ZEXE: Enabling Decentralized Private Computation

Sean Bowe
sean@z.cash
Zcash

Alessandro Chiesa
alexch@berkeley.edu
UC Berkeley

Matthew Green
mgreen@cs.jhu.edu
Johns Hopkins University

Ian Miers
imiers@cs.jhu.edu
Cornell Tech

Pratyush Mishra
pratyush@berkeley.edu
UC Berkeley

Howard Wu
howardwu@berkeley.edu
UC Berkeley

October 8, 2018

Abstract

Ledger-based systems that enable rich applications often suffer from two limitations. First, validating a transaction requires re-executing the state transition that it attests to. Second, transactions not only reveal which application had a state transition but also reveal the application's internal state. Unfortunately, expensive re-execution and lack of privacy rule out many use cases.

We design, implement, and evaluate ZEXE, a ledger-based system where users can execute offline computations and subsequently produce transactions, attesting to the correctness of these computations, that satisfy two main properties. First, transactions hide all information about the offline computations. Second, transactions can be validated by anyone in constant time, regardless of the offline computation.

The core of ZEXE is a protocol for a new cryptographic primitive that we introduce, *decentralized private computation* (DPC). The security guarantees of DPC are concisely expressed via an ideal functionality, which our protocol provably achieves. In order to achieve an efficient implementation of our protocol, we leverage tools in the area of cryptographic proofs, including succinct zero knowledge proofs and recursive proof composition. Overall, transactions in ZEXE are 968 bytes *regardless of the offline computation*, and generating them takes less than 2 minutes plus a time that grows with the offline computation.

To facilitate real-world deployments, ZEXE also provides support for delegating the process of producing a transaction to an untrusted worker, and support for threshold transactions and blind transactions.

Keywords: decentralized computation; zero knowledge proofs; succinct arguments

Contents

1 Introduction

Distributed ledgers are a mechanism that maintains data across a distributed system while ensuring that every party has the same view of the data, even in the presence of corrupted parties. Ledgers can provide an indisputable history of all "events" logged in a system, thereby offering a mechanism for multiple parties to collaborate with minimal trust (any party can ensure the system's integrity by auditing history). Interest in distributed ledgers has soared recently, catalyzed by their use in cryptocurrencies (peer-to-peer payment systems) and by their potential as a foundation for new forms of financial systems, governance, and data sharing. In this work we study two limitations of ledgers, one about *privacy* and the other about *scalability*.

The privacy problem. The main strength of distributed ledgers is also their main weakness: *the history of all events is available for anyone to read*. This severely limits a direct application of distributed ledgers.

For example, in ledger-based payment systems such as Bitcoin [Nak09], every payment transaction reveals the payment's sender, receiver, and amount. This not only reveals private financial details of individuals and businesses using the system,[1] but also violates fungibility, a fundamental economic property of money. This lack of privacy becomes more severe in smart contract systems like Ethereum [Woo17], wherein transactions not only contain payment details, but also embed function calls to specific applications. In these systems, every application's internal state is public, and so is the history of function calls associated to it.

This problem has motivated prior work to find ways to achieve meaningful privacy guarantees on ledgers. For example, the Zerocash protocol [BCG$^+$14] provides privacy-preserving payments, and Hawk [KMS$^+$16] enables general state transitions with data privacy, that is, an application's data is hidden from third parties.

However, all prior work is limited to hiding the inputs and outputs of a state transition but not *which* transition function is being executed. That is, prior work achieves *data privacy* but not also *function privacy*. In systems with a single transition function this is not a concern.[2] In systems with multiple transition functions, however, this leakage is problematic. For example, Ethereum currently supports thousands of separate ERC-20 "token" contracts [Eth18], each representing a distinct currency on the Ethereum ledger; even if these contracts each individually adopted a protocol such as Zerocash to hide details about token payments, the corresponding transactions would still reveal *which* token was being exchanged. Moreover, the leakage of this information would substantially reduce the anonymity set of those payments.

The re-execution problem. Public auditability in the aforementioned systems (and many others) is achieved via direct verification of state transitions. This creates two problems. First, validating a transaction involves re-executing the associated computation and so, to discourage denial-of-service attacks whereby users send transactions that take a long time to validate, current systems introduce mechanisms such as *gas* to make users pay more for longer computations. Second, even with such mechanisms, validating an expensive transaction may simply not be economically profitable, a problem known as the "Verifier's Dilemma" [LTKS15]. These problems have resulted in Bitcoin forks [Bit15] and Ethereum attacks [Eth16].

In sum, there is a dire need for techniques that facilitate the use of distributed ledgers for rich applications, without compromising privacy (of data or functions) or relying on unnecessary re-executions. Prior works only partially address this need, as discussed in Section 1.2 below.

[1]Even if payments merely contain *addresses* rather than, say, social security numbers, much information about individuals and businesses can be gleaned by analyzing the flow of money over time between addresses [RH11, RS13, AKR$^+$13, MPJ$^+$13, SMZ14, KGC$^+$17]. There are even companies that offer analytics services on the information stored on ledgers [Ell13, Cha14].

[2]For example, in Zerocash the single transition function is the one governing cash flow of a single currency.

1.1 Our contributions

We design, implement, and evaluate ZEXE (*Zero knowledge EXEcution*), a ledger-based system that enables users to execute offline computations and subsequently produce publicly-verifiable transactions that attest to the correctness of these offline executions. ZEXE simultaneously provides two main security properties.

- **Privacy:** *a transaction reveals no information about the offline computation, except (an upper bound on) the number of consumed inputs and created outputs.*[3] One cannot link together multiple transactions by the same user or involving related computations, nor selectively censor transactions based on such information.
- **Succinctness:** *a transaction can be validated in time that is independent of the cost of the offline computation whose correctness it attests to.* Since all transactions are equally cheap to validate (they are all indistinguishable), there is no "Verifier's Dilemma" nor a need for mechanisms such as *gas*.

ZEXE also offers rich functionality, as offline computations in ZEXE can be used to realize state transitions of multiple applications (such as tokens, elections, markets) simultaneously running atop the *same* ledger. The users participating in applications to do not have to trust, or even know of, one another. ZEXE supports this functionality by exposing a simple, yet powerful, *shared execution environment* with the following properties.

- **Extensibility:** users may execute arbitrary functions of their choice, without seeking anyone's permission.
- **Isolation:** functions of malicious users cannot interfere with the computations and data of honest users.
- **Inter-process communication:** functions may exchange data with one another.

DPC schemes. The technical core of ZEXE is a protocol for a new cryptographic primitive that we introduce, *decentralized private computation* (DPC), a new approach to performing computations on a ledger. Informally, DPC supports a simple, yet expressive, programming model in which units of data, which we call *records*, contain within them scripts (arbitrary programs) that determine under what conditions they can be first created and then consumed. The rules that dictate how these programs interact can be viewed as a "nano-kernel" that provides a shared execution environment upon which to build applications. From a technical perspective, DPC can be viewed as extending Zerocash [BCG+14] to the foregoing programming model, while still providing strong privacy guarantees, not only within a single application (which is a straightforward extension) but also across multiple co-existing applications (which requires new ideas that we discuss later on). The security guarantees of DPC are concisely expressed via an ideal functionality, which our protocol provably achieves.

Techniques for efficient implementation. We devise a set of techniques to achieve an efficient implementation of our DPC protocol, by drawing upon recent advances in zero knowledge succinct cryptographic proofs (namely, zkSNARKs) and in recursive proof composition (proofs attesting to the validity of other proofs).

Overall, transactions in ZEXE with two input records and two output records are 968 bytes and can be verified in tens of milliseconds, *regardless of the offline computation*; generating these transactions takes less than 2 minutes plus a time that grows with the offline computation (inevitably so). This implementation is achieved in a modular fashion via a collection of Rust libraries (see Fig. 15), in which the top-level one is `libzexe`. Our implementation also supports transactions with *any* number m of input records and n of output records; transactions size in this case is $32m + 32n + 840$ bytes (the transaction stores the serial number of each input record and the commitment of each output record).

[3] One can fix the number of inputs and outputs (say, fix both to 2), or carefully consider side channels that could arise from revealing bounds on the number of inputs and outputs.

Delegating transactions. While verifying succinct cryptographic proofs is cheap, producing them can be expensive. As the offline computation grows, the (time and space) cost of producing a cryptographic proof of its correctness also grows, which could become infeasible for a user.

To address this problem, we further obtain *delegable DPC*. The user communicates to an untrusted worker worker details about the desired transaction, then the worker produces the transaction, and finally the user authorizes it via a cheap computation (and in a way that does not violate indistinguishability of transactions). This feature is particularly relevant for prospective real-world deployments, because it enables support for weak devices, such as mobile phones or hardware tokens.

In fact, our delegable DPC protocol also extends to support *threshold transactions*, which can be used to improve operational security, and also to support *blind transactions*, which can be used to realize lottery tickets for applications such as micropayments.

All of these extensions are also part of our Rust library `libzexe`.

A perspective on costs. Zexe provides tolerable efficiency but is by no means a lightweight construction. We have, after all, set ambitious goals: *data/function privacy and succinctness, for a rich functionality, in a threat model that requires security against all efficient adversaries*. Relaxing any of these goals (assuming rational adversaries or hardware enclaves, or compromising on privacy) will lead to more efficient approaches.

In light of the foregoing ambitious goals, we have, in our opinion, managed to achieve excellent transaction sizes (less than a kilobyte) and transaction verification times (tens of milliseconds).

The main undesirable cost in our system is, not surprisingly, the cost to generate the cryptographic proofs to include in transactions. We have managed to keep this cost to under 2 minutes plus a cost that grows with the offline computation, which is similar to what prior systems have achieved for *more limited* functionalities [BCG$^+$14]. But we are optimistic that this cost can be significantly reduced via more careful engineering.

1.2 Related work

Avoiding naive re-execution. TrueBit [TR17], Plasma [PB17], and Arbitrum [KGC$^+$18] avoid naive re-execution by having users report the results of their computations *without* any cryptographic proofs, and instead putting in place incentive mechanisms wherein others can challenge reported results. The user and challenger engage in a so-called *refereed game* [FK97, CRR11, CRR13, JSST16, Rei16], mediated by a smart contract acting as the referee, that efficiently determines which of the two was "telling the truth". In contrast, in this work correctness of computation is ensured by cryptography, regardless of any economic motives; we thus protect against all efficient adversaries rather than merely all rational and efficient ones. Also, unlike our DPC scheme, the above works do not provide formal guarantees of strong privacy (challengers must be able to re-execute the computation leading to a result and in particular must know its potentially private inputs).

Private payments. Zerocash [BCG$^+$14], building on earlier work [MGGR13], showed how to use distributed ledgers to achieve payment systems with strong privacy guarantees. Informally, users encrypt payment details and prove their validity, without disclosing what the payment details are. The Zerocash protocol, with some modifications, is now commercially deployed in several currencies, including Zcash [ZCa15]. In Zerocash, however, there is no support for scripting, that is, specifying small programs that dictate how funds can be spent. Even more so, in Zerocash there is no support for complex financial logic, and more generally for programming arbitrary state transitions like in smart contract systems such as Ethereum.

Privacy beyond payments. Hawk [KMS$^+$16], combining ideas from Zerocash and the notion of an evaluator-prover for multi-party computation, enables parties to conduct offline computations and then report their results via cryptographic proofs. The privacy guarantee that Hawk achieves, known as transactional privacy, protects the private inputs used in a computation (directly so from the proofs' zero knowledge property) but does not protect the information of *which* computation was carried out. That said, we view Hawk as complementary to our work: a user in our system could in particular be a semi-trusted manager that administers a multi-party computation and generates a transaction about its output. The privacy guarantees provided in this work would hide *which* computation was carried out offline.

MPC with ledgers. Several works [ADMM14b, ADMM14a, KMB15, KB16, BKM17] have applied ledgers to obtain secure multi-party protocols that have security properties that are difficult to achieve otherwise, such as *fairness*. These approaches are complementary to our work, as any set of parties wishing to jointly compute a certain function via one of these protocols could run the protocol "under" our DPC scheme in such a way that third parties would not learn any information that such a multi-party computation is happening.

Hardware enclaves. Ekiden [CZK$^+$18] is a ledger-based system that uses hardware enclaves, such as Intel Software Guard Extensions [MAB$^+$13], to achieve various integrity and privacy goals for smart contracts. Beyond ledgers, several systems explore privacy goals in distributed systems by leveraging hardware enclaves; see for example M2R [DSC$^+$15], VC3 [SCF$^+$15], and Opaque [ZDB$^+$17]. All of these works are able to efficiently support rich and complex computations. In this work, we make no use of hardware enclaves, and instead rely entirely on cryptography. This means that on the one hand our performance overheads are more severe, while on the other hand we protect against a richer class of adversaries (all efficient ones).

2 Techniques

We summarize the main ideas behind our contributions. Our goal is to design a ledger-based system in which transactions attest to offline computations while simultaneously providing *privacy* and *succinctness*.

We begin by noting that privacy is the "harder" of the two goals, since there is a straightforward folklore approach that provides succinctness alone: each user accompanies the result reported in a transaction with a succinct cryptographic proof (i.e., a SNARK) attesting to the result's correctness. Others who validate the transaction can then simply verify the cryptographic proof, and do not have to re-execute the computation. In light of this, we shall first discuss how to achieve privacy, and then how to additionally achieve succinctness.

The rest of this section is organized as follows. In Sections 2.1 and 2.2 we explain why achieving privacy in our setting is challenging. In Section 2.3 we introduce the shared execution environment that we consider, and in Section 2.4 we introduce *decentralized private computation* (DPC), a cryptographic primitive that securely realizes it. In Section 2.5 we describe how we turn our ideas into an efficient implementation.

2.1 Achieving privacy for a single arbitrary function

Zerocash [BCG$^+$14] is a protocol that achieves privacy for a specific functionality, namely, *value transfers within a single currency*. Therefore, it is natural to consider what happens if we extend Zerocash from this special case to the general case of a *single arbitrary function* that is known in advance to everybody.

Sketch of Zerocash. Money in Zerocash is represented via *coins*. The commitment of a coin is published on the ledger when the coin is created, and its serial number is published when the coin is consumed. Each transaction on the ledger attests that some "old" coins were consumed in order to create some "new" coins: it contains the serial numbers of the consumed coins, commitments of the created coins, and a zero knowledge proof attesting that the serial numbers belong to coins created in the past (without identifying which ones), and that the commitments contain new coins of the same total value. A transaction is private because it only reveals how many coins were consumed and how many were created, but no other information (each coin's value and owner address remain hidden). Also, revealing a coin's serial number ensures that a coin cannot be consumed more than once (the same serial number would appear twice). In sum, data in Zerocash corresponds to coin values, and state transitions are the single invariant that monetary value is preserved.

Extending to an arbitrary function. One way to extend Zerocash to a single arbitrary function Φ (known in advance to everybody) is to think of a coin as a *record* that stores some arbitrary data *payload*, rather than just some integer value. The commitment of a record would then be published on the ledger when the record is created, and its unique serial number would be published when the record is consumed. A transaction would then contain serial numbers of consumed records, commitments of created records, and a proof attesting that invoking the function Φ on (the payload of) the old records produces (the payload of) the new records.

Data privacy holds because the ledger merely stores each record's commitment (and its serial number once consumed), and transactions only reveal that some number of old records were consumed in order to create some number of new records in a way that is consistent with Φ. Function privacy also holds but for trivial reasons: Φ is known in advance to everybody, and every transaction is about computations of Φ.

Note that Zerocash is indeed a special case of the above: it corresponds to fixing Φ to the particular (and publicly known) choice of a function $\Phi_{\$}$ that governs value transfers within a single currency.

However the foregoing protocol supports only a single hard-coded function Φ, while instead we want to enable users to select their own functions, as we discuss next.

2.2 Difficulties with achieving privacy for user-defined functions

We want to enable users to execute functions of their choice concurrently on the same ledger, while maintaining function privacy and without seeking prior permission from anyone. That is, when preparing a transaction, a user may pick *any* function Φ of his choice for creating new records by consuming some old records.

This alone *can* be achieved via the approach sketched in Section 2.1 by fixing a single function that is *universal*, and then interpreting data payloads as user-defined functions that are provided as inputs. Indeed, zero knowledge would ensure function privacy in this case. However merely allowing users to define their own functions does *not* by itself yield meaningful functionality, as we explain next.

The problem: malicious functions. Users could devise functions to attack or disrupt other users' functions and data, so that a particular user would not know whether to trust records created by other users; indeed, due to function privacy, he would not know what functions were used to create those records. For example, suppose that we wanted to realize the special case of value transfers within a single currency (i.e., Zerocash). One may believe that it would suffice to instruct users to pick the function $\Phi_{\$}$ (or similar). But this does *not* work: a user receiving a record claiming to contain, say, 1 unit of currency does not know if this record was created via the function $\Phi_{\$}$ from other such records and so on. A malicious user could have used a different function to create that record, for example, one that illegally "mints" records that appear valid to $\Phi_{\$}$. More generally, the lack of any enforced rules about how user-defined functions can interact precludes productive cooperation between users that do not trust one another. We stress that this challenge arises specifically due to function privacy, because if the function that created (the commitment of) a record was public knowledge, users could decide for themselves if records they receive were generated by a "good" functions.

One way to address the foregoing problem would be to augment records with an attribute that *must* equal the identity of the function that created them, and then impose the restriction that in a valid transaction only records created by the same function may participate. This new attribute is never revealed on the ledger (just like a record's payload), and the zero knowledge proof is tasked with ensuring that records participating in the same transaction are all of the same "type". This approach now *does* suffice to realize value transfers within a single currency, by letting users select the function $\Phi_{\$}$. More generally, this approach generalizes that in Section 2.1, and can be viewed as running multiple segregated "virtual ledgers" each with a fixed function. Function privacy holds because one cannot tell if a transaction belongs to one virtual ledger or another.

The problem: limited functionality. The foregoing forbids any inter-process communication, and so one cannot realize even simple functionalities like transferring value between different currencies on the same ledger. This crude time sharing of the ledger is too limiting.

2.3 The records nano-kernel: a minimalist shared execution environment

The approaches in Section 2.2 lie at opposite extremes: unrestricted inter-process interactions cannot realize even basic applications such as a single currency, while complete process segregation is too limiting.

Balancing these extremes requires a shared execution environment, namely an *operating system*, that manages user-defined functions: it provides process isolation, determines data ownership, handles inter-process communication, and so on. Overall, processes must be able to concurrently share a ledger, without violating the integrity or confidentiality of one another.

However, function privacy (one of our goals) dictates that user-defined functions are hidden, which means that an operating system cannot be maintained publicly atop the ledger (as in current smart contract systems) but, instead, must be part of the statement proved in zero knowledge. This is unfortunate because designing an operating system that governs interactions across user-defined functions within a zero knowledge proof is not only a colossal design challenge but also entails many arbitrary design choices that we should not have to take.

In light of the above, we choose to take the following approach: we formulate a *minimalist* shared execution environment that imposes simple, yet expressive, rules on how records may interact. This execution environment can be viewed as a "nano-kernel" that manages records, and can be summarized as follows.

> **The records nano-kernel (RNK):** In addition to a data payload, a record $\mathbf{r}$ will now contain two user-defined functions, or more precisely two user-defined *predicates* (boolean functions). These are a *birth* predicate Φ_{b}, which is executed when $\mathbf{r}$ is created, and a *death* predicate Φ_{d}, which is executed when $\mathbf{r}$ is consumed. By suitably programming $\mathbf{r}$'s birth and death predicates, a user fully dictates the conditions under which $\mathbf{r}$ can be first created and later consumed.

As before, a transaction in the ledger attests that some old records were consumed in order to create new records. However, now it also attests that the old records' death predicates and the new records' birth predicates were all simultaneously satisfied when given a certain common input. This input is the transaction's *local data*, which includes: (a) every record's contents (such as its payload and the identity of its predicates); (b) a piece of shared memory that is publicly revealed, called *transaction memorandum*; (c) a piece of shared memory that is kept hidden, called *auxiliary input*; and (d) other construction specifics. See Fig. 3.

In this way, *each predicate can individually decide if the local data is valid according to its own logic*. For example, a record can protect itself from other records that contain "bad" birth or death predicates, because the record's predicates could refuse to accept when they detect (from reading the local data) that they are in a transaction with records having bad predicates. At the same time, a record can interact with other records in the same transaction when its predicates decide to accept, providing the flexibility that we seek.

In Example 2.1 below we illustrate this flexibility via a simple application. More generally, "applications" arise as emergent systems on top of carefully designed sets of predicates. Records appearing in the same transaction can use the shared memory to perform joint computation and communication. In fact, computation and communication may span across multiple transactions, by storing suitable intermediate state/message data in record payloads, or by publishing that data in transactions memoranda (as plaintext or ciphertext as needed). Thus, an arbitrary computation may be either conducted within the scope of a single transaction, or instead it could take place across many transactions, e.g., by thinking of transactions as realizing state transitions, with consumed records viewed as data loaded from memory, and created records viewed as data stored back to memory. These perspectives demonstrate that the records nano-kernel is a powerful model for computation.

Example 2.1 (user-defined tokens)**.** We briefly explain how to use the records nano-kernel to realize an application wherein users can define new tokens and then exchange them according to desired exchange rates.

We consider records whose payloads encode two pieces of information: a token identifier id and a value v. We fix the birth predicate in all such records to be a function $\Phi_{\mathsf{b}}^{\star}$ that is responsible for creating the initial supply of a new token, and then subsequently conserving the value of the token across all transactions. In more detail, $\Phi_{\mathsf{b}}^{\star}$ can be invoked in one of two modes. In *mint mode*, given as input a desired initial supply v_0, $\Phi_{\mathsf{b}}^{\star}$ deterministically derives a fresh unique identifier id for a new token (this can be done via information unique to a transaction) and stores the pair (id, v_0) in a newly created *genesis record*. In *conserve mode*, $\Phi_{\mathsf{b}}^{\star}$ inspects all records in a transaction whose birth predicates equal to $\Phi_{\mathsf{b}}^{\star}$ and whose token identifiers equal the identifier of the current record, and ensures that among these the token values are conserved.

Users can program death predicates of records to enforce conditions on how tokens can be consumed, e.g., by realizing *conditional exchanges* with other counter-parties. Suppose that wants wishes to exchange 100 units of a token id_1 for 50 units of another token id_2, but does not have a counter-party for the exchange. She creates a record $\mathbf{r}$ with 100 units of id_1 whose death predicate enforces that any transaction consuming $\mathbf{r}$ must also create another record, consumable by Alice, with 50 units of id_1. She then publishes out of band information about $\mathbf{r}$, and anyone can subsequently claim it by creating a transaction doing the exchange.

2.4 Decentralized private computation

A new cryptographic primitive. We introduce a new cryptographic primitive called *decentralized private computation* (DPC) schemes, which capture the notion of a ledger-based system where privacy-preserving transactions attest to offline computations that follow the records nano-kernel. See Section 3 for the definition of DPC schemes, including the ideal functionality that we use to express security.

We construct a DPC scheme in Section 4, and prove it secure in Appendix A. We take Zerocash [BCG$^+$14] as a starting point, and then extend the protocol to support the records nano-kernel and also to facilitate proving security in the simulation paradigm relative to an ideal functionality (rather than via a collection of separate game-based definitions as in [BCG$^+$14]). Below we sketch the construction.

Construction sketch. Each transaction in the ledger consumes some old records and creates new records in a manner that is consistent with the records nano-kernel. To ensure privacy, a transaction only contains serial numbers of the consumed records, commitments of the created records, and a zero knowledge proof attesting that there exist records consistent with this information (and with the records nano-kernel). All commitments on the ledger are collected in a Merkle tree, which facilitates efficiently proving that a commitment appears on the ledger (by proving in zero knowledge the knowledge of a suitable authentication path). All serial numbers on the ledger are collected in a list that cannot contain duplicates. This implies that a record cannot be consumed twice because the same serial number is revealed each time a record is consumed. See Fig. 1.

The record data structure is summarized in Fig. 2. Each record is associated to an *address public key*, which is a commitment to a seed for a pseudorandom function acting as the corresponding *address secret key*; addresses determine ownership of records, and in particular consuming a record requires knowing its secret key. A *record* consists of an address public key, a data payload, a birth predicate, a death predicate, and a serial number nonce; a *record commitment* is a commitment to all of these attributes. The *serial number* of a record is the evaluation of a pseudorandom function, whose seed is the secret key for the record's address public key, evaluated at the record's serial number nonce. A record's commitment and serial number, which appear on the ledger when the record is created and consumed, reveal *no* information about the record attributes. This follows from the hiding properties of the commitment, and the pseudorandom properties of the serial number. The derivation of a record's serial number ensures that a user can create a record for another in such a way that its serial number is fully determined and yet cannot be predicted without knowing the other user's secret key.

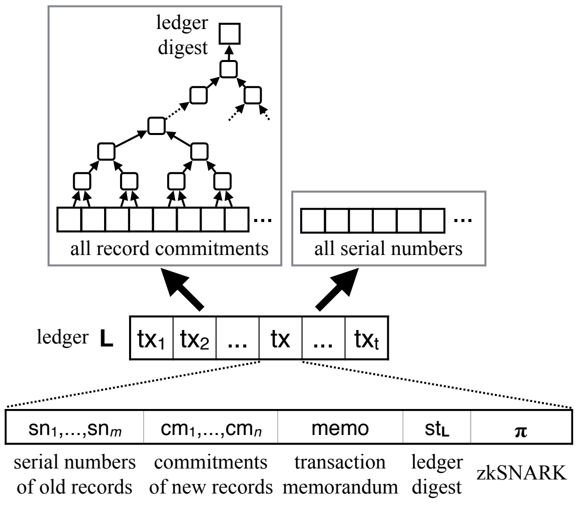

Figure 1: Construction of a transaction.

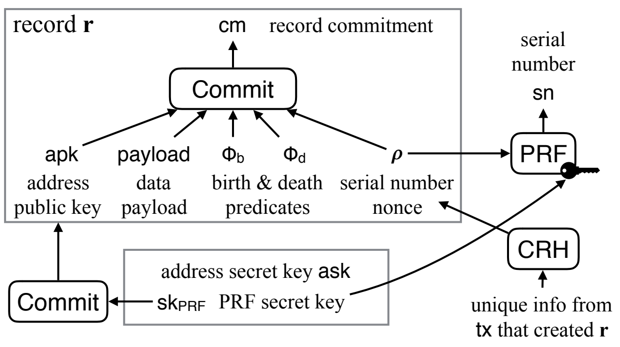

Figure 2: Construction of a record.

In order to produce a transaction, a user selects some previously-created records to consume, assembles some new records to create (including their payloads and predicates), and decides on other aspects of the local data such as the transaction memorandum (shared memory seen by all predicates and published on the ledger) and the auxiliary input (shared memory seen by all predicates but not published on the ledger); see Fig. 3. If the user knows the secret keys of the records to consume and if all relevant predicates are satisfied (death predicates of old records and birth predicates of new predicates), then the user can produce a zero knowledge proof to append to the transaction. See Fig. 4 for a summary of the NP statement being proved.

In sum, a transaction only reveals the number of consumed records and number of created records, as well as any data that was deliberately revealed in the transaction memorandum (possibly nothing).[4]

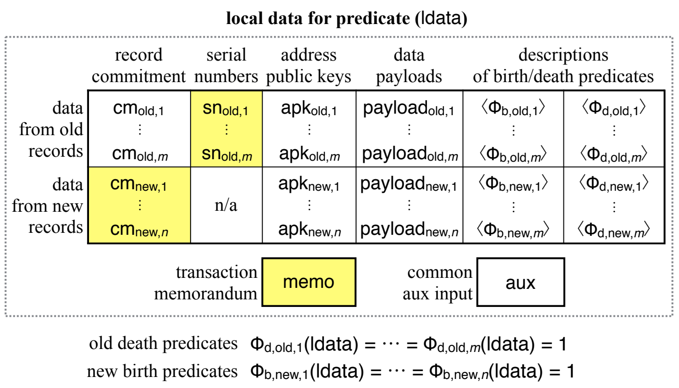
local data for predicate (ldata)
record commitment
serial numbers
address public keys
data payloads
descriptions of birth/death predicates
data from old records
data from new records
$\mathsf{cm}_{\mathsf{old},1}$
$\mathsf{cm}_{\mathsf{old},m}$
$\mathsf{sn}_{\mathsf{old},1}$
$\mathsf{sn}_{\mathsf{old},m}$
$\mathsf{apk}_{\mathsf{old},1}$
$\mathsf{apk}_{\mathsf{old},m}$
$\mathsf{payload}_{\mathsf{old},1}$
$\mathsf{payload}_{\mathsf{old},m}$
$\langle\Phi_{\mathsf{b},\mathsf{old},1}\rangle$
$\langle\Phi_{\mathsf{b},\mathsf{old},m}\rangle$
$\langle\Phi_{\mathsf{d},\mathsf{old},1}\rangle$
$\langle\Phi_{\mathsf{d},\mathsf{old},m}\rangle$
$\mathsf{cm}_{\mathsf{new},1}$
$\mathsf{cm}_{\mathsf{new},n}$
n/a
$\mathsf{apk}_{\mathsf{new},1}$
$\mathsf{apk}_{\mathsf{new},n}$
$\mathsf{payload}_{\mathsf{new},1}$
$\mathsf{payload}_{\mathsf{new},n}$
$\langle\Phi_{\mathsf{b},\mathsf{new},1}\rangle$
$\langle\Phi_{\mathsf{b},\mathsf{new},m}\rangle$
$\langle\Phi_{\mathsf{d},\mathsf{new},1}\rangle$
$\langle\Phi_{\mathsf{d},\mathsf{new},m}\rangle$
transaction memorandum
memo
common aux input
aux
old death predicates $\Phi_{\mathsf{d},\mathsf{old},1}(\mathsf{ldata}) = \cdots = \Phi_{\mathsf{d},\mathsf{old},m}(\mathsf{ldata}) = 1$
new birth predicates $\Phi_{\mathsf{b},\mathsf{new},1}(\mathsf{ldata}) = \cdots = \Phi_{\mathsf{b},\mathsf{new},n}(\mathsf{ldata}) = 1$

Figure 3: Predicates receive local data.

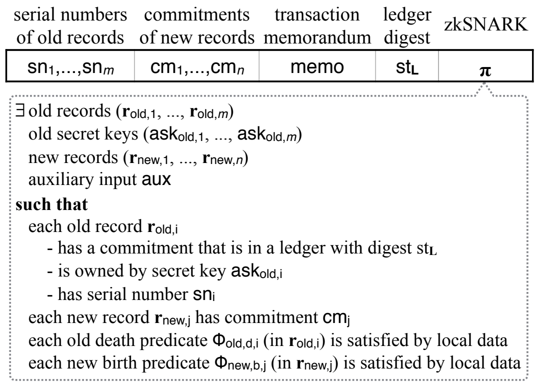
serial numbers of old records
commitments of new records
transaction memorandum
ledger digest
zkSNARK
$\mathsf{sn}_1,\ldots,\mathsf{sn}_m$
$\mathsf{cm}_1,\ldots,\mathsf{cm}_n$
memo
st_L
π
$\exists$ old records $(\mathbf{r}_{\mathsf{old},1}, \ldots, \mathbf{r}_{\mathsf{old},m})$
old secret keys $(\mathsf{ask}_{\mathsf{old},1}, \ldots, \mathsf{ask}_{\mathsf{old},m})$
new records $(\mathbf{r}_{\mathsf{new},1}, \ldots, \mathbf{r}_{\mathsf{new},n})$
auxiliary input aux
such that
each old record $\mathbf{r}_{\mathsf{old},i}$
- has a commitment that is in a ledger with digest st_L
- is owned by secret key $\mathsf{ask}_{\mathsf{old},i}$
- has serial number sn_i
each new record $\mathbf{r}_{\mathsf{new},j}$ has commitment cm_j
each old death predicate $\Phi_{\mathsf{old},\mathsf{d},i}$ (in $\mathbf{r}_{\mathsf{old},i}$) is satisfied by local data
each new birth predicate $\Phi_{\mathsf{new},\mathsf{b},j}$ (in $\mathbf{r}_{\mathsf{new},j}$) is satisfied by local data

Figure 4: The execute statement.

Achieving succinctness. Our discussions so far have focused on achieving (data and function) privacy. However, we also want to achieve succinctness, namely, that a transaction can be validated in "constant time". This follows from a straightforward modification: we take the protocol that we have designed so far and use a zero knowledge *succinct* argument rather than just any zero knowledge proof. Indeed, the NP statement being proved (summarized in Fig. 4) involves attesting the satisfiability of all (old) death and (new) birth predicates, and thus we need to ensure that verifying the corresponding proof can be done in time that does not depend on the complexity of these predicates. While turning this idea into an efficient implementation requires more ideas (as we discuss in Section 2.5), the foregoing modification suffices from a theoretical point of view.

Delegation to an untrusted worker. In our DPC scheme, a user must produce, and include in the transaction, a zero knowledge succinct argument that, among other things, attests that death predicates of consumed records are satisfied and, similarly, that birth predicates of created records are satisfied. This implies that the cost of creating a transaction grows with the complexity (and number of) predicates involved in the transaction. Such a cost can quickly become infeasible for weak devices such as mobile phones or hardware tokens.

We address this problem by enabling a user to *delegate* to an untrusted worker, such as a remote server, the computation that produces a transaction. This notion, which we call a *delegable DPC scheme*, empowers weak devices to produce transactions that they otherwise could not have produced on their own.

The basic idea is to augment address keys in such a way that the secret information needed to produce the cryptographic proof is separate from the secret information needed to authorize a transaction containing that proof. Thus, the user can communicate to the worker the secrets necessary to generate a cryptographic proof, while retaining the remaining secrets for authorizing this (and future) transactions. In particular, the worker has no way to produce valid transactions that have not been authorized by the user.

[4]By supporting the use of dummy records, we can in fact ensure that only *upper bounds* on the foregoing numbers are revealed.

We use randomizable signatures to achieve the foregoing functionality, without violating either privacy or succinctness. Informally, we modify a record's serial number to be an unlinkable randomization of (part of) the record's address public key, and a user's authorization of a transaction consists of signing the instance and proof relative to every randomized key (i.e., serial number) in that transaction. See Section 5 for details.

2.5 Achieving an efficient implementation

Our system ZEXE (*Zero knowledge EXEcution*) provides an implementation of two constructions: our "plain" DPC protocol, and its extension to a delegable DPC protocol. Achieving efficiency in our system required overcoming several challenges. Below we highlight some of these challenges, and explain how we addressed them; see Sections 7 and 8 for details. The discussions below equally apply to both types of DPC protocols.

Avoiding the cost of universality. The NP statement that we need to prove involves checking user-defined predicates, so it must support arbitrary computations that are not fixed in advance. However, state-of-the-art zkSNARKs for universal computations rely on expensive tools [BCG$^+$13, BCTV14, WSR$^+$15, BCTV17].

We address this problem by relying on one layer of *recursive proof composition* [Val08, BCCT13]. Instead of tasking the NP statement with directly checking user-defined predicates, we only task it with checking *succinct proofs* attesting to this. Checking these *inner* succinct proofs is a (relatively) inexpensive computation that is fixed for *all* predicates, which can be "hardcoded" in the statement. Since the single *outer* succinct proof produced does not reveal information about the inner succinct proofs attesting to predicates' satisfiability (thanks to zero knowledge), the inner succinct proofs do *not* have to hide what predicate was checked, so they can be for NP statements *tailored* to the computations of particular user-defined predicates.

A bespoke recursion. Recursive proof composition has been empirically demonstrated for pairing-based SNARKs [BCTV17]. We thus focus our attention on these, and explain the challenges that arise in our setting. Recall that if we instantiate a SNARK's pairing via an elliptic curve E defined over a prime field $\mathbb{F}_q$ and having a subgroup of prime order r, then (a) the SNARK supports NP statements expressed as arithmetic circuits over $\mathbb{F}_r$, while (b) proof verification involves arithmetic operations over $\mathbb{F}_q$. Being part of the NP statement, the SNARK verifier must also be expressed as an arithmetic circuit over $\mathbb{F}_r$, which is problematic because the verifier's "native" operations are over $\mathbb{F}_q$. Simulating $\mathbb{F}_q$ operations via $\mathbb{F}_r$ operations is expensive, and picking E such that $q = r$ is impossible [BCTV17]. Prior work thus uses *multiple* curves [BCTV17]: a two-cycle of pairing-friendly elliptic curves, that is, two prime-order curves E_1 and E_2 such that the prime size of one's base field is the prime order of the other's group, and orchestrating SNARKs based on these so that fields "match up". However, known cycles are inefficient at 128 bits of security [BCTV17, CCW18].

We address this problem by noting that we merely need "a proof of a proof", and thus, instead of relying on a cycle, we can use the Cocks–Pinch method [FST10] to set up a bounded recursion [BCTV17]. First we pick a pairing-friendly elliptic curve that not only is suitable for 128 bits of security according to standard considerations but, moreover, is compatible with efficient SNARK provers in *both* levels of the recursion. Namely, letting p be the prime order of the base field and r the prime order of the group, we need that *both* $\mathbb{F}_r$ and $\mathbb{F}_p$ have multiplicative subgroups whose orders are large powers of 2. The condition on $\mathbb{F}_r$ ensures efficient proving for SNARKs over this curve, while the condition on $\mathbb{F}_p$ ensures efficient proving for SNARKs that verify proofs over this curve. In light of the above, we select a curve E_{BLS} from the Barreto–Lynn–Scott (BLS) family [BLS02, CLN11] with embedding degree 12. This family not only enables parameters that conservatively achieve 128 bits of security, but also enjoys properties that facilitate very efficient implementation [AFK$^+$12]. We ensure that both $\mathbb{F}_r$ and $\mathbb{F}_p$ have multiplicative subgroups of order 2^α for $\alpha \geq 40$, by a suitable condition on the parameter of the BLS family.

Next we use the Cocks–Pinch method to pick a pairing-friendly elliptic curve E_{CP} over a field $\mathbb{F}_q$ such that the curve group $E_{\mathsf{CP}}(\mathbb{F}_q)$ contains a subgroup of prime order p (the size of E_{BLS}'s base field). Since the method outputs a prime q that has about $2\times$ more bits than the desired p, and in turn p has about $1.5\times$ more bits than r (due to properties of the BLS family), we only need E_{CP} to have embedding degree of 6 in order to achieve 128 bits of security (as determined from the guidelines in [FST10]).

In sum, a SNARK over E_{BLS} is used to generate proofs of predicates' satisfiability; after that a zkSNARK over E_{CP} is used to generate proofs that these prior proofs are valid along with the remaining NP statement's checks. The matching fields between the two curves ensure that the former proofs can be efficiently verified.

Minimizing operations over E_{CP}. While the curve E_{CP} facilitates efficient checking of SNARK proofs over E_{BLS}, operations on it are at least $2\times$ more costly (in time and space) than operations over E_{BLS}, simply because E_{CP}'s base field is twice the size of E_{BLS}'s base field. This makes checks in the NP relation $\mathcal{R}_{\mathsf{e}}$ that are not related to proof checking unnecessarily expensive.

To avoid this, we split $\mathcal{R}_{\mathsf{e}}$ into two NP relations, $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$. The latter is responsible only for verifying proofs of predicates' satisfaction, while the former is responsible for all other checks. We minimize the number of E_{CP} operations by proving satisfaction of $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$ with zkSNARKs over E_{BLS} and E_{CP} respectively. A transaction now includes both proofs.

Optimizing the NP statement. We note that the remaining NP statement's checks can themselves be quite expensive, as they range from verifying authentication paths in a Merkle tree to verifying commitment openings, and from evaluating pseudorandom functions to evaluating collision resistant functions. Prior work realizing similar collections of checks required upwards of *four million gates* [BCG$^+$14] to express such checks. This not only resulted in high latencies for producing transactions (several minutes) but also resulted in large public parameters for the system (hundreds of megabytes).

Commitments and collision-resistant hashing can be expressed as very efficient arithmetic circuits if one opts for Pedersen-type constructions over suitable Edwards elliptic curves (and techniques derived from these ideas are now part of deployed systems [HBHW18]). To achieve this, we pick two Edwards curves, $E_{\mathsf{Ed/BLS}}$ over the field $\mathbb{F}_r$ (thereby matching the group order of E_{BLS}), and $E_{\mathsf{Ed/CP}}$ over the field $\mathbb{F}_p$ (thereby matching the group order of E_{CP}). This allows to realise very efficient circuits for various primitives used in our NP relations, including commitments, collision-resistant hashing, and randomizable signatures. Overall, we obtain highly optimized realizations of all checks in Fig. 4.

3 Definition of decentralized private computation schemes

We define *decentralized private computation* (DPC) schemes, a cryptographic primitive in which parties with access to an ideal append-only ledger execute computations offline and subsequently post privacy-preserving, publicly-verifiable transactions that attest to the correctness of these offline executions. This primitive generalizes prior notions [BCG+14] that were limited to proving correctness of simple financial invariants.

Below we introduce the data structures, interface, and security requirements for a DPC scheme: Section 3.1 describes the main data structures of a DPC scheme, Section 3.2 defines the syntax of the DPC algorithms, and finally in Section 3.3 we describe the security requirements for DPC schemes via an ideal functionality. We note that our definition of DPC schemes focuses on (correctness and) privacy, because we leave succinctness as a separate efficiency goal that easily follows from suitable building blocks (see Remark 4.1).

3.1 Data structures

In a DPC scheme there are three main data structures: *records*, *transactions*, and the *ledger*.

Records. A *record*, denoted by the symbol $\mathbf{r}$, is a data structure representing a unit of data. Records can be created or consumed, and these events denote state changes in the system. For example, in a currency application, records store units of the currency, and state changes represent the flow of units in that currency.

In more detail, a record $\mathbf{r}$ has the following attributes (see Fig. 5): (a) a *commitment* cm, which binds together all other attributes of $\mathbf{r}$ while hiding all information about them; (b) an *address public key* apk, which specifies the record's owner; (c) a *payload* $\mathsf{payload}$ containing arbitrary application-dependent information; (d) a *birth predicate* Φ_{b} that must be satisfied when $\mathbf{r}$ is created; (e) a *death predicate* Φ_{d} that must be satisfied when $\mathbf{r}$ is consumed; and (f) other construction-specific information. Both Φ_{b} and Φ_{d} are arbitrary non-deterministic boolean-valued functions. The payload $\mathsf{payload}$ contains a designated subfield $\mathsf{isDummy}$ which denotes whether $\mathbf{r}$ is dummy or not.

Informally, the "life" of a (non-dummy) record $\mathbf{r}$ is marked by two events: *birth* and *death*. The record $\mathbf{r}$ is *born* (or is *created*) when its commitment cm is posted to the ledger as part of a transaction. Then the record $\mathbf{r}$ *dies* (or is *consumed*) when its serial number sn appears on the ledger as part of a later transaction. At each of these times (birth or death) the corresponding predicate (Φ_{b} or Φ_{d}) must be satisfied. Dummy records, on the other hand, can be created freely, but consuming them requires satisfaction of their death predicates. The purpose of dummy records is solely to enable the creation of new non-dummy records.

To consume $\mathbf{r}$, one must also know the address secret key ask corresponding to $\mathbf{r}$'s address public key apk because the serial number sn to be revealed can only be computed from $\mathbf{r}$ and ask. The ledger forbids the same serial number to appear more than once, so that: (a) a record cannot be consumed twice because it is associated to exactly one serial number; (b) others cannot prevent one from consuming a record because it is computationally infeasible to create two distinct records that share the same serial number sn but have distinct commitments cm and cm'.

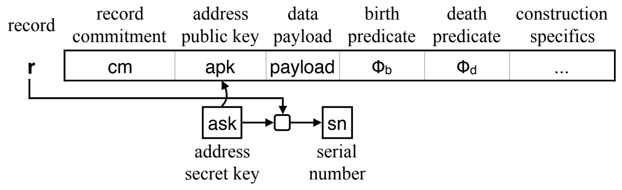

Figure 5: Diagram of a record.

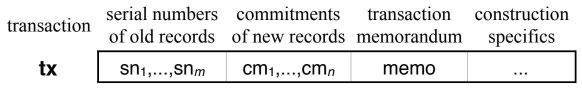

Figure 6: Diagram of a transaction.

Transactions. A transaction, denoted by the symbol tx, is a data structure representing a state change that involves the consumption and creation of records (see Fig. 6). It is a tuple $([\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo}, \star)$ where (a) $[\mathsf{sn}_i]_1^m$ is the list of serial numbers of the m old records, (b) $[\mathsf{cm}_j]_1^n$ is the list of commitments of the n new records, (c) memo is an arbitrary string associated with the transaction, and (d) $\star$ is other construction-specific information. The transaction tx reveals only the following information about old and new records: (i) the old records' serial numbers; (ii) the new records' commitments; and (iii) the fact that the death predicates of all consumed records and birth predicates of all new records were satisfied.

Anyone can assemble a transaction and append it to the ledger, provided that it is "valid" in the sense that (all records are well-formed and) the death predicates of any consumed records and the birth predicates of any created records are satisfied. Note that all transactions reveal the number of old records (m) and the number of new records (n), but not how many of these were dummy or not.

Ledger. We consider a model where all parties have access to an append-only ledger, denoted $\mathbf{L}$, that stores all published transactions. Our definitions (and constructions) are *agnostic* to how this ledger is realized (e.g., the ledger may be centrally managed or a distributed protocol). When an algorithm needs to interact with the ledger, we specify $\mathbf{L}$ in the algorithm's superscript. The ledger exposes the following interface.

- $\mathbf{L}.\mathsf{Len}$: Return the number of transactions currently on the ledger.
- $\mathbf{L}.\mathsf{Push}(\mathsf{tx})$: Append a (valid) transaction tx to the ledger.
- $\mathbf{L}.\mathsf{Digest} \to \mathsf{st}_{\mathbf{L}}$: Return a (short) digest of the current state of the ledger.
- $\mathbf{L}.\mathsf{ValidateDigest}(\mathsf{st}_{\mathbf{L}}) \to b$: Check that $\mathsf{st}_{\mathbf{L}}$ is a valid digest for some (past) ledger state.
- $\mathbf{L}.\mathsf{Contains}(\mathsf{tx}) \to b$: Determine if tx (or a subcomponent thereof) appears on the ledger or not.
- $\mathbf{L}.\mathsf{Prove}(\mathsf{tx}) \to \mathbb{w}_{\mathbf{L}}$: If a transaction tx (or a subcomponent thereof) appears on the ledger, return a proof of membership $\mathbb{w}_{\mathbf{L}}$ for it. If there are duplicates, return a proof for the lexicographically first one.
- $\mathbf{L}.\mathsf{Verify}(\mathsf{st}_{\mathbf{L}}, \mathsf{tx}, \mathbb{w}_{\mathbf{L}}) \to b$: Check that $\mathbb{w}_{\mathbf{L}}$ certifies that tx (or a subcomponent thereof) is in a ledger with digest $\mathsf{st}_{\mathbf{L}}$.

We stress that only "valid" transactions can be appended to the ledger. While the full definition of a valid transaction is implementation dependent, in all cases it must be that the commitments and serial numbers in a transaction (including any appearing in the $\star$ field of a transaction) do not already appear on the ledger.

3.2 Algorithms

A DPC scheme is a tuple of algorithms (some of which may read information from $\mathbf{L}$):

$$\mathsf{DPC} = (\mathsf{Setup}, \mathsf{GenAddress}, \mathsf{Execute}^{\mathbf{L}}, \mathsf{Verify}^{\mathbf{L}}) \ .$$

The syntax and semantics of these algorithms are informally described below.

Setup: $\mathsf{DPC.Setup}(1^\lambda) \to \mathsf{pp}$.

On input a security parameter 1^λ, $\mathsf{DPC.Setup}$ outputs public parameters pp for the system. A trusted party runs this algorithm once and then publishes its output; afterwards the trusted party is not needed anymore.

For some constructions, the trusted party can be replaced by an efficient multiparty computation that securely realizes the $\mathsf{DPC.Setup}$ algorithm (see [BCG+15, ZCa16, BGG17] for how this has been done in some systems); in other constructions, the trusted party may not be needed, as the public parameters may simply consist of a random string of a certain length.

Create address: $\mathsf{DPC.GenAddress}(\mathsf{pp}, \mathsf{meta}) \to (\mathsf{apk}, \mathsf{ask})$.

On input public parameters pp and address metadata meta, DPC.GenAddress outputs an address key pair (apk, ask). The address metadata is bound to apk so that it is computationally infeasible to create two equal address public keys with different metadata. Any user may run this algorithm to create an address key pair. Each record is bound to an address public key, and the corresponding secret key is used to consume it.

Execute: Any user may invoke DPC.Execute to consume records and create new ones.

$$\mathsf{DPC.Execute}^{\mathbf{L}}\left(\begin{array}{ll} \text{public parameters} & \mathsf{pp} \\ \text{old records} & [\mathbf{r}_i]_1^m \\ \text{old address secret keys} & [\mathsf{ask}_i]_1^m \\ \text{new address public keys} & [\mathsf{apk}_j]_1^n \\ \text{new record payloads} & [\mathsf{payload}_j]_1^n \\ \text{new record birth predicates} & [\Phi_{\mathsf{b},j}]_1^n \\ \text{new record death predicates} & [\Phi_{\mathsf{d},j}]_1^n \\ \text{auxiliary predicate input} & \mathsf{aux} \\ \text{transaction memorandum} & \mathsf{memo} \end{array}\right) \to \left(\begin{array}{ll} \text{new records} & [\mathbf{r}_j]_1^n \\ \text{transaction} & \mathsf{tx} \end{array}\right).$$

Given as input a list of old records $[\mathbf{r}_i]_1^m$ with corresponding secret keys $[\mathsf{ask}_i]_1^m$, attributes for new records, private auxiliary input aux to birth and death predicates of new and old records respectively,[5] and an arbitrary transaction memorandum memo, DPC.Execute produces new records $[\mathbf{r}_j]_1^n$ and a transaction tx. The transaction attests that the input records' death predicates and the output records' birth predicates are all satisfied. The user subsequently pushes tx to the ledger by invoking $\mathbf{L}.\mathsf{Push}(\mathsf{tx})$.

Verify: $\mathsf{DPC.Verify}^{\mathbf{L}}(\mathsf{pp}, \mathsf{tx}) \to b$.

On input public parameters pp and a transaction tx, and given oracle access to the ledger $\mathbf{L}$, DPC.Verify outputs a bit b denoting whether the transaction tx is valid relative to the ledger $\mathbf{L}$.

3.3 Security

Informally, a DPC scheme achieves the following security goals.

- *Execution correctness.* Malicious parties cannot create valid transactions if the death predicate of some consumed record or the birth predicate of some created record is not satisfied.
- *Execution privacy.* Transactions reveal only the information revealed in the memorandum field, a bound on the number of consumed records, and a bound on the number of created records.[6] All other information is hidden, including the payloads and predicates of all involved records. For example, putting aside the information revealed in the memorandum (which is arbitrary), one cannot link a transaction that consumes a record with the prior transaction that created it.
- *Consumability.* Every record can be consumed at least once and at most once by parties that know its secrets. Thus, a malicious party cannot create two valid records for another party such that only one of them can be consumed. (This captures security against "faerie-gold" attacks [HBHW18].)
- *Transaction non-malleability.* Malicious parties cannot modify a transaction "in flight" to the ledger.

[5]In addition to the "global" auxiliary input aux, each predicate may also take as input a "local" auxiliary input that is not (necessarily) shared with other predicates. For simplicity, we make these local inputs implicit.

[6]And any information implied by knowing that the birth (resp., death) predicates of consumed (resp., created) records are satisfied.

Formally, we prove *standalone* security against *static corruptions*, in a model where every party has private anonymous channels to all other parties [IKOS06].[7] (In Appendix C we discuss how to prove security under composition and against adaptive corruptions.) In more detail, we capture security of a DPC scheme via a *simulation-based* security definition that is akin to UC security [Can01], but restricted to a single execution.

Definition 3.1. *A DPC scheme* DPC *is* **secure** *if for every efficient real-world adversary* $\mathcal{A}$ *there exists an efficient ideal-world simulator* $\mathcal{S}_{\mathcal{A}}$ *such that for every efficient environment* $\mathcal{E}$ *the following are computationally indistinguishable:*

- *the output of* $\mathcal{E}$ *when interacting with the adversary* $\mathcal{A}$ *in a real-world execution of* DPC *in a model where parties can communicate with other parties via private anonymous channels; and*
- *the output of* $\mathcal{E}$ *when interacting with the simulator* $\mathcal{S}_{\mathcal{A}}$ *in an ideal-world execution with the ideal functionality* $\mathcal{F}_{\mathrm{DPC}}$ *specified in Fig. 7 (and further described below).*

We describe the data structures used by the ideal functionality $\mathcal{F}_{\mathrm{DPC}}$, the internal state of $\mathcal{F}_{\mathrm{DPC}}$, and the interface offered by $\mathcal{F}_{\mathrm{DPC}}$ to parties in the ideal-world execution.

Ideal data structures. The ideal functionality $\mathcal{F}_{\mathrm{DPC}}$ uses ideal counterparts of a DPC scheme's data structures. An *address public key* apk denotes the owner of an *ideal record* $\mathbb{r}$, which is a tuple $(\mathsf{cm}, \mathsf{apk}, \mathsf{payload}, \Phi_{\mathsf{b}}, \Phi_{\mathsf{d}})$, where cm is its commitment, apk is its address public key, $\mathsf{payload}$ is its payload, and Φ_{b} and Φ_{d} are its birth and death predicates. The record is also associated with a unique identifier (or *serial number*) sn. We require that apk, cm, and sn are "globally unique"; this means that there cannot be two different ideal records $\mathbb{r}$ and $\mathbb{r}'$ having the same commitments or serial numbers.

The distribution of these components is specified by the simulator $\mathcal{S}$ as follows. Before the ideal execution begins, $\mathcal{S}$ specifies three functions $(\mathsf{SampleAddrPk}, \mathsf{SampleCm}, \mathsf{SampleSn})$ that, on input a random string, sample $(\mathsf{apk}, \mathsf{cm}, \mathsf{sn})$ respectively. When $\mathcal{F}_{\mathrm{DPC}}$ needs to sample one of these, it invokes the respective functions. (Note that $\mathcal{F}_{\mathrm{DPC}}$ cannot directly ask $\mathcal{S}$ to sample these because that would reveal to $\mathcal{S}$ when an honest party was invoking $\mathcal{F}_{\mathrm{DPC}}.\mathsf{GenAddress}$ or $\mathcal{F}_{\mathrm{DPC}}.\mathsf{Execute}$, and we cannot afford this leakage.)

Internal state. The ideal functionality $\mathcal{F}_{\mathrm{DPC}}$ maintains several internal tables.

- Addr, which maps an address public key to metadata.
- $\mathsf{AddrUsers}$, which maps an address public key to the set of parties that are authorized to use it.
- $\mathsf{Records}$, which maps a record's commitment to that record's information (address public key, payload, birth predicate, and death predicates).
- $\mathsf{RecUsers}$, which maps a record's commitment to the set of parties that are authorized to consume it. Note that, for a record $\mathbb{r}$, the set $\mathsf{RecUsers}[\mathbb{r}.\mathsf{cm}]$ can be different from the set in $\mathsf{AddrUsers}[\mathbb{r}.\mathsf{apk}]$, but a party $\mathcal{P}$ has to be in both sets to consume $\mathbb{r}$.
- $\mathsf{SerialNumbers}$, which maps a record's commitment to that record's (unique) serial number.
- State, which maps a record's commitment to that record's state, either `alive` or `dead`.

Ideal algorithms. The ideal functionality $\mathcal{F}_{\mathrm{DPC}}$ provides the following interface to parties.

- *Address generation:* $\mathcal{F}_{\mathrm{DPC}}.\mathsf{GenAddress}$ outputs a new address public key apk.
- *Execution:* $\mathcal{F}_{\mathrm{DPC}}.\mathsf{Execute}$ performs an execution that consumes old records and creates new records. All parties are notified that an execution has occurred, and learn the serial numbers of input records, commitments of output records, and the transaction memorandum memo. Concurrent $\mathcal{F}_{\mathrm{DPC}}.\mathsf{Execute}$ calls are serialized arbitrarily.
- *Record consumption authorization:* $\mathcal{F}_{\mathrm{DPC}}.\mathsf{ShareRecord}$ allows a party $\mathcal{P}$ to authorize another party $\mathcal{P}'$ to consume a record $\mathbb{r}$ (provided that $\mathcal{P}'$ is also authorized to use $\mathbb{r}$'s address public key).

[7] Parties can, e.g., use these channels to communicate the contents of newly created records to other parties.

Operation of honest parties. In both the real and ideal executions, the environment $\mathcal{E}$ can send instructions to honest parties. These instructions can be one of $\mathsf{GenAddress}$, $\mathsf{Execute}$, or $\mathsf{ShareRecord}$. In the real world honest parties translate these instructions into corresponding invocations of DPC algorithms (or messages sent via private anonymous channels as in the case of $\mathsf{ShareRecord}$), while in the ideal world they translate them into corresponding invocations of $\mathcal{F}_{\mathrm{DPC}}$ algorithms. In both worlds, honest parties immediately invoke $\mathsf{ShareRecord}$ on records obtained from an $\mathsf{Execute}$ instruction. Finally, in the ideal world, when invoking $\mathcal{F}_{\mathrm{DPC}}$, honest parties do not provide any inputs marked as optional; instead, they let $\mathcal{F}_{\mathrm{DPC}}$ sample these.

Intuition. We explain how $\mathcal{F}_{\mathrm{DPC}}$ enforces the informal security notions described at this section's beginning.

- *Execution correctness.* $\mathcal{F}_{\mathrm{DPC}}.\mathsf{Execute}$ ensures that the death predicates of consumed records and birth predicates of created records are satisfied by the local data. Note that each predicate receives its own position as input so that it knows to which record in the local data it belongs.
- *Execution privacy.* Transactions contain serial numbers $[\mathsf{sn}_i]_1^m$ of consumed records, commitments $[\mathsf{cm}_j]_1^n$ of created records, and a memorandum memo. Serial numbers and commitments are sampled via $\mathsf{SampleSn}$ and $\mathsf{SampleCm}$, so they are independent of the contents of any record, and thus reveal no information about them. Transactions thus reveal no information (beyond what is contained in memo).
- *Consumability.* From the point of view of $\mathcal{F}_{\mathrm{DPC}}$, two records are different if and only if they have different commitments. In such a case, both records can be consumed as long as their death predicates are satisfied. If a DPC scheme realizes $\mathcal{F}_{\mathrm{DPC}}$, then it must satisfy this same requirement: if two valid records have distinct commitments, then they must both be consumable.
- *Transaction non-malleability.* The adversary has no power to modify the inputs to, or output of, an honest party's invocation of $\mathcal{F}_{\mathrm{DPC}}.\mathsf{Execute}$.

```
$\mathcal{F}_{\text{DPC}}.\mathsf{GenAddress}[\mathcal{P}]\begin{pmatrix}\text{address metadata} & \mathsf{meta}\\ \text{(optional) address public key} & \mathsf{apk}\end{pmatrix}$
1. Sample randomness $r$ for generating address public key.
2. If $\mathsf{apk} = \bot$ then $\mathsf{apk} \leftarrow \mathsf{SampleAddrPk}(r)$.
3. Check that $\mathsf{apk}$ is **unique**: $\mathsf{Addr}[\mathsf{apk}] = \bot$.
4. Set $\mathsf{Addr}[\mathsf{apk}] := (\mathsf{meta}, r)$.
5. If $\mathcal{P}$ is corrupted: set $S$ to be the set of corrupted parties.
6. If $\mathcal{P}$ is honest: set $S := \{\mathcal{P}\}$.
7. Set $\mathsf{AddrUsers}[\mathsf{apk}] := \mathsf{AddrUsers}[\mathsf{apk}] \cup S$.
8. **Send to $\mathcal{P}$:** address public key $\mathsf{apk}$.
```

```
$\mathcal{F}_{\text{DPC}}.\mathsf{ShareRecord}[\mathcal{P}]\begin{pmatrix}\text{record} & \mathtt{r}\\ \text{recipient party} & \mathcal{P}'\end{pmatrix}$
1. If $\mathsf{Records}[\mathtt{r}.\mathsf{cm}] \neq \bot$:
   (a) Check that $\mathcal{P} \in \mathsf{RecUsers}[\mathtt{r}.\mathsf{cm}]$.
   (b) Retrieve $((\mathsf{cm}, \mathsf{apk}, \mathsf{payload}, \Phi_{\mathsf{b}}, \Phi_{\mathsf{d}}), r) := \mathsf{Records}[\mathtt{r}.\mathsf{cm}]$.
2. If $\mathcal{P}'$ is corrupted: set $S$ to be the set of corrupted parties.
3. If $\mathcal{P}'$ is honest: set $S := \{\mathcal{P}'\}$.
4. Set $\mathsf{RecUsers}[\mathtt{r}.\mathsf{cm}] := \mathsf{RecUsers}[\mathtt{r}.\mathsf{cm}] \cup S$.
5. If $\mathcal{P}$ is honest and $\mathcal{P}'$ isn't, **Send to $\mathcal{P}'$:** $(\mathtt{RecordAuth}, (\mathtt{r}, r))$.
6. Else, **Send to $\mathcal{P}'$:** $(\mathtt{RecordAuth}, \mathtt{r})$.
```

```
$\mathcal{F}_{\text{DPC}}.\mathsf{Execute}[\mathcal{P}]\begin{pmatrix}\text{old records} & [\mathtt{r}_i]_1^m\\ \text{old address metadata} & [\mathsf{meta}_i]_1^m\\ \text{(optional) old serial numbers} & [\mathsf{sn}_i]_1^m\\ \text{(optional) new record commitments} & [\mathsf{cm}_j]_1^n\\ \text{new address public keys} & [\mathsf{apk}_j]_1^n\\ \text{new record payloads} & [\mathsf{payload}_j]_1^n\\ \text{new record birth predicates} & [\Phi_{\mathsf{b},j}]_1^n\\ \text{new record death predicates} & [\Phi_{\mathsf{d},j}]_1^n\\ \text{auxiliary predicate input} & \mathsf{aux}\\ \text{transaction memorandum} & \mathsf{memo}\end{pmatrix}$
1. For each $i \in \{1, \dots, m\}$:
   (a) Sample randomness $r_i$.
   (b) If $\mathsf{sn}_i = \bot$ then generate **serial number**: $\mathsf{sn}_i \leftarrow \mathsf{SampleSn}(r_i)$.
   (c) Check that $\mathsf{sn}_i$ is **unique**: $\mathsf{SerialNumbers}[\mathsf{sn}_i] = \bot$.
2. For each $j \in \{1, \dots, n\}$:
   (a) Sample randomness $r_j$.
   (b) If $\mathsf{cm}_j = \bot$ then generate **commitment**: $\mathsf{cm}_j \leftarrow \mathsf{SampleCm}(r_j)$.
   (c) Check that $\mathsf{cm}_j$ is **unique**: $\mathsf{Records}[\mathsf{cm}_j] = \bot$.
   (d) **Construct record**: $\mathtt{r}_j := (\mathsf{cm}_j, \mathsf{apk}_j, \mathsf{payload}_j, \Phi_{\mathsf{b},j}, \Phi_{\mathsf{d},j})$.
3. Define the local data $\mathsf{ldata} := ([\mathtt{r}_i]_1^m, [\mathsf{sn}_i]_1^m, [\mathsf{meta}_i]_1^m, [\mathtt{r}_j]_1^n, \mathsf{aux}, \mathsf{memo})$.
4. For each $i \in \{1, \dots, m\}$:
   (a) Parse $\mathtt{r}_i$ as $(\mathsf{cm}_i, \mathsf{apk}_i, \mathsf{payload}_i, \Phi_{\mathsf{b},i}, \Phi_{\mathsf{d},i})$.
   (b) Check that, for some randomness $r_i$, **old record** $\mathtt{r}_i$ **exists**: $((\mathsf{apk}_i, \mathsf{payload}_i, \Phi_{\mathsf{b},i}, \Phi_{\mathsf{d},i}), r_i) = \mathsf{Records}[\mathsf{cm}_i]$.
   (c) Check that $\mathcal{P}$ **is authorized to use** $\mathsf{apk}_i$: $\mathcal{P} \in \mathsf{AddrUsers}[\mathsf{apk}_i]$.
   (d) If $\mathsf{payload}_i.\mathsf{isDummy} = 0$:
         i. Check that **record is unconsumed**: $\mathsf{State}[\mathtt{r}_i] = \mathtt{alive}$.
        ii. Check that $\mathcal{P}$ **is authorized to consume** $\mathtt{r}_i$: $\mathcal{P} \in \mathsf{RecUsers}[\mathsf{cm}_i]$.
       iii. Check that $\mathcal{P}$ **is authorized to use** $\mathsf{apk}_i$: $\mathcal{P} \in \mathsf{AddrUsers}[\mathsf{apk}_i]$.
   (e) Check that **death predicate is satisfied**: $\Phi_{\mathsf{d},i}(i \| \mathsf{ldata}) = 1$.
   (f) **Mark it as consumed**: $\mathsf{State}[\mathsf{cm}_i] := \mathtt{dead}$.
5. For each $j \in \{1, \dots, n\}$:
   (a) Check that **birth predicate is satisfied**: $\Phi_{\mathsf{b},j}(j \| \mathsf{ldata}) = 1$.
   (b) **Insert new record $\mathbf{r}_j$**: $\mathsf{Records}[\mathsf{cm}_j] := ((\mathsf{apk}_j, \mathsf{payload}_j, \Phi_{\mathsf{b},j}, \Phi_{\mathsf{d},j}), r_j)$.
   (c) **Mark new record as unconsumed**: $\mathsf{State}[\mathsf{cm}_j] := \mathtt{alive}$.
6. **Send to $\mathcal{P}$:** $([\mathtt{r}_j]_1^n)$.
7. **Send to all parties:** $(\mathtt{Execute}, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo})$.
```

Figure 7: Ideal functionality $\mathcal{F}_{\text{DPC}}$ of a DPC scheme.

4 Construction of decentralized private computation schemes

We describe our construction of a DPC scheme. In Section 4.1 we introduce the building blocks that we use, and in Section 4.2 we describe each algorithm in the scheme. The security proof is provided in Appendix A. We also describe some extensions of our construction, in functionality and in security, in Appendix C.

4.1 Building blocks

CRHs. A collision-resistant hash function $\mathsf{CRH} = (\mathsf{Setup}, \mathsf{Eval})$ works as follows.

- *Setup:* on input a security parameter, $\mathsf{CRH.Setup}$ samples public parameters $\mathsf{pp}_{\mathsf{CRH}}$.
- *Hashing:* on input public parameters $\mathsf{pp}_{\mathsf{CRH}}$ and message m, $\mathsf{CRH.Eval}$ outputs a short hash h of m.

Given public parameters $\mathsf{pp}_{\mathsf{CRH}} \leftarrow \mathsf{CRH.Setup}(1^\lambda)$, it is computationally infeasible to find distinct inputs x and y such that $\mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, x) = \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, y)$.

PRFs. A pseudorandom function family $\mathsf{PRF} = \{\mathsf{PRF}_x \colon \{0,1\}^* \to \{0,1\}^{O(|x|)}\}_x$, where x denotes the seed, is computationally indistinguishable from a random function family.

Trapdoor commitments. A trapdoor commitment scheme $\mathsf{TCM} = (\mathsf{Setup}, \mathsf{Commit})$ enables a party to generate a (perfectly) hiding and (computationally) binding commitment to a given message.

- *Setup:* on input a security parameter, $\mathsf{TCM.Setup}$ samples public parameters $\mathsf{pp}_{\mathsf{TCM}}$.
- *Commitment:* on input public parameters $\mathsf{pp}_{\mathsf{TCM}}$, message m, and randomness r_{cm}, $\mathsf{TCM.Commit}$ outputs a commitment cm to m.

Auxiliary algorithms enable producing a trapdoor, and using it to open a commitment, originally to an empty string, to an arbitrary message. These algorithms are used only in the proof of security, and so we introduce them there (see Appendix A).

NIZKs. Non-interactive zero knowledge arguments of knowledge enable a party, known as the *prover*, to convince another party, known as the *verifier*, about knowledge of the witness for an NP statement without revealing any information about the witness (besides what is already implied by the statement being true). This primitive is a tuple $\mathsf{NIZK} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ with the following syntax.

- *Setup:* on input a security parameter and the specification of an NP relation $\mathcal{R}$, $\mathsf{NIZK.Setup}$ outputs a set of public parameters $\mathsf{pp}_{\mathsf{NIZK}}$ (also known as a *common reference string*).
- *Proving:* on input $\mathsf{pp}_{\mathsf{NIZK}}$ and an instance-witness pair $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, $\mathsf{NIZK.Prove}$ outputs a proof π.
- *Verifying:* on input $\mathsf{pp}_{\mathsf{NIZK}}$, instance $\mathbb{x}$, and proof π, $\mathsf{NIZK.Verify}$ outputs a decision bit.

Completeness states that honestly generated proofs make the verifier accept; *(computational) proof of knowledge* states that if the verifier accepts a proof for an instance then the prover "knows" a witness for it; and *perfect zero knowledge* states that honestly generated proofs can be perfectly simulated, when given a trapdoor to the public parameters. In fact, we require a strong form of (computational) proof of knowledge known as *simulation-extractability*, which states that proofs continue to be proofs of knowledge even when the adversary has seen prior simulated proofs. For more details, see [Sah99, DDO$^+$01, Gro06].

Remark 4.1. If NIZK is additionally *succinct* (i.e., it is a simulation-extractable zkSNARK) then the DPC scheme constructed in this section is also *succinct*. This is the case in our implementation; see Section 8.

4.2 Algorithms

Pseudocode for our construction of a DPC scheme is in Fig. 8. The construction involves invoking zero knowledge proofs for the NP relation $\mathcal{R}_{\mathsf{e}}$ described in Fig. 9. The text below is a summary of the construction.

System setup. $\mathsf{DPC.Setup}$ is a wrapper around the setup algorithms of cryptographic building blocks. It invokes $\mathsf{TCM.Setup}$, $\mathsf{CRH.Setup}$, and $\mathsf{NIZK.Setup}$ to obtain trapdoor commitment public parameters $\mathsf{pp_{TCM}}$, CRH public parameters $\mathsf{pp_{CRH}}$, and NIZK public parameters for the NP relation $\mathcal{R}_\mathsf{e}$ (see Fig. 9). It then outputs $\mathsf{pp} := (\mathsf{pp_{TCM}}, \mathsf{pp_{CRH}}, \mathsf{pp_e})$.

Address creation. $\mathsf{DPC.GenAddress}$, on input address metadata meta, constructs an address key pair as follows. The address secret key ask is a tuple $(\mathsf{sk_{PRF}}, \mathsf{meta}, r_\mathsf{pk})$ consisting of a secret key $\mathsf{sk_{PRF}}$ for the pseudorandom function PRF, the address metadata meta, and commitment randomness r_pk. The address public key apk is a commitment to $\mathsf{sk_{PRF}} \| \mathsf{meta}$ with randomness r_pk. Thus, apk binds together $\mathsf{sk_{PRF}}$ and meta, while simultaneously hiding all information about them.

Execution. $\mathsf{DPC.Execute}$ produces a transaction attesting that some old records $[\mathbf{r}_i]_1^m$ were consumed and some new records $[\mathbf{r}_j]_1^n$ were created, and that their death and birth predicates were satisfied. First, $\mathsf{DPC.Execute}$ computes a ledger membership witness and serial number for every old record. Then, $\mathsf{DPC.Execute}$ invokes the following auxiliary function to create record commitments for the new records.

```
DPC.ConstructRecord(pp, apk, payload, $\Phi_\mathsf{b}, \Phi_\mathsf{d}, \rho$) $\to$ ($\mathbf{r}$, cm)
1. Sample new commitment **randomness** $r$.
2. Construct new record **commitment**: $\mathsf{cm} \leftarrow \mathsf{TCM.Commit}(\mathsf{pp_{TCM}}, \mathsf{apk} \| \mathsf{payload} \| \Phi_\mathsf{b,} \| \Phi_\mathsf{d,} \| \rho;\ r)$.
3. Construct new **record** $\mathbf{r} := \left( \begin{array}{cccccc} \text{address public key} & \mathsf{apk} & \text{payload} & \mathsf{payload} & \text{comm. rand.} & r \\ \text{serial number nonce} & \rho & \text{predicates} & (\Phi_\mathsf{b}, \Phi_\mathsf{d}) & \text{commitment} & \mathsf{cm} \end{array} \right)$.
4. Output $(\mathbf{r}, \mathsf{cm})$.
```

Information about all records, secret addresses of old records, the desired transaction memorandum memo, and desired auxiliary predicate input aux are collected into the local data ldata (see Fig. 9).

Finally, $\mathsf{DPC.Execute}$ produces a proof that all records are well-formed and that several conditions hold.

- *Old records are properly consumed*, namely, for every old record $\mathbf{r}_i \in [\mathbf{r}_i]_1^m$:
  - (if $\mathbf{r}_i$ is not dummy) $\mathbf{r}_i$ *exists*, demonstrated by checking a ledger membership witness for $\mathbf{r}_i$'s commitment;
  - $\mathbf{r}_i$ *has not been consumed*, demonstrated by publishing $\mathbf{r}_i$'s serial number sn_i;
  - $\mathbf{r}_i$*'s death predicate* $\Phi_{\mathsf{d},i}$ *is satisfied*, demonstrated by checking that $\Phi_{\mathsf{d},i}(i \| \mathsf{ldata}) = 1$.
- *New records are property created*, namely, for every new record $\mathbf{r}_j \in [\mathbf{r}_j]_1^n$:
  - $\mathbf{r}_j$*'s serial number is unique*, achieved by generating the nonce ρ_j as $\mathsf{CRH.Eval}(\mathsf{pp_{CRH}}, j \| \mathsf{sn}_1 \| \dots \| \mathsf{sn}_m)$;
  - $\mathbf{r}_j$*'s birth predicate* $\Phi_{\mathsf{b},j}$ *is satisfied*, demonstrated by checking that $\Phi_{\mathsf{b},j}(j \| \mathsf{ldata}) = 1$.

The serial number sn of a record $\mathbf{r}$ relative to an address secret key $\mathsf{ask} = (\mathsf{sk_{PRF}}, \mathsf{meta}, r_\mathsf{pk})$ is derived by evaluating PRF at $\mathbf{r}$'s serial number nonce ρ with seed $\mathsf{sk_{PRF}}$. This ensures that sn is pseudorandom even to a party that knows all of $\mathbf{r}$ but not ask (e.g., to a party that created the record for some other party). Note that each predicate receives its own position as input so that it knows to which record in the local data it belongs.

```
$\mathsf{DPC.Setup}$
Input: security parameter $1^\lambda$
Output: public parameters $\mathsf{pp}$

1. Generate trapdoor commitment parameters:
   $\mathsf{pp_{TCM}} \leftarrow \mathsf{TCM.Setup}(1^\lambda)$.
2. Generate CRH parameters: $\mathsf{pp_{CRH}} \leftarrow \mathsf{CRH.Setup}(1^\lambda)$.
3. Generate NIZK parameters for $\mathcal{R}_\mathsf{e}$ (see Figure 9):
   $\mathsf{pp_e} \leftarrow \mathsf{NIZK.Setup}(1^\lambda, \mathcal{R}_\mathsf{e})$.
4. Output $\mathsf{pp} := (\mathsf{pp_{TCM}}, \mathsf{pp_{CRH}}, \mathsf{pp_e})$.
```

```
$\mathsf{DPC.GenAddress}$
Input: public parameters $\mathsf{pp}$ and address metadata $\mathsf{meta}$
Output: address key pair $(\mathsf{apk}, \mathsf{ask})$

1. Sample secret key $\mathsf{sk_{PRF}}$ for pseudorandom function $\mathsf{PRF}$.
2. Sample randomness $r_\mathsf{pk}$ for commitment scheme $\mathsf{TCM}$.
3. Set address public key
   $\mathsf{apk} := \mathsf{TCM.Commit}(\mathsf{pp_{TCM}}, \mathsf{sk_{PRF}} \| \mathsf{meta};\ r_\mathsf{pk})$.
4. Set address secret key $\mathsf{ask} := (\mathsf{sk_{PRF}}, \mathsf{meta}, r_\mathsf{pk})$.
5. Output $(\mathsf{apk}, \mathsf{ask})$.
```

```
$\mathsf{DPC.Execute}^\mathbf{L}$
Input:
  • public parameters $\mathsf{pp}$
  • old $\begin{cases} \text{records } [\mathbf{r}_i]_1^m \\ \text{address secret keys } [\mathsf{ask}_i]_1^m \end{cases}$
  • new $\begin{cases} \text{address public keys } [\mathsf{apk}_j]_1^n \\ \text{record payloads } [\mathsf{payload}_j]_1^n \\ \text{record birth predicates } [\Phi_{\mathsf{b},j}]_1^n \\ \text{record death predicates } [\Phi_{\mathsf{d},j}]_1^n \end{cases}$
  • auxiliary predicate input $\mathsf{aux}$
  • transaction memorandum $\mathsf{memo}$
Output: new records $[\mathbf{r}_j]_1^n$ and transaction $\mathsf{tx}$

1. For each $i \in \{1, \ldots, m\}$, process the $i$-th old record as follows:
   (a) Parse old record $\mathbf{r}_i$ as $\left(\begin{array}{cccccc} \text{address public key} & \mathsf{apk}_i & \text{payload} & \mathsf{payload}_i & \text{comm. rand.} & r_i \\ \text{serial number nonce} & \rho_i & \text{predicates} & (\Phi_{\mathsf{b},i}, \Phi_{\mathsf{d},i}) & \text{commitment} & \mathsf{cm}_i \end{array}\right)$.
   (b) If $\mathsf{payload}_i.\mathsf{isDummy} = 1$, set ledger membership witness $\mathbb{w}_{\mathbf{L},i} := \bot$.
       If $\mathsf{payload}_i.\mathsf{isDummy} = 0$, compute ledger membership witness for commitment: $\mathbb{w}_{\mathbf{L},i} \leftarrow \mathbf{L}.\mathsf{Prove}(\mathsf{cm}_i)$.
   (c) Parse address secret key $\mathsf{ask}_i$ as $(\mathsf{sk}_{\mathsf{PRF},i}, \mathsf{meta}_i, r_{\mathsf{pk},i})$.
   (d) Compute serial number: $\mathsf{sn}_i \leftarrow \mathsf{PRF}_{\mathsf{sk}_{\mathsf{PRF},i}}(\rho_i)$.
2. For each $j \in \{1, \ldots, n\}$, construct the $j$-th new record as follows:
   (a) Compute serial number nonce: $\rho_j := \mathsf{CRH.Eval}(\mathsf{pp_{CRH}}, j \| \mathsf{sn}_1 \| \ldots \| \mathsf{sn}_m)$.
   (b) Construct new record: $(\mathbf{r}_j, \mathsf{cm}_j) \leftarrow \mathsf{DPC.ConstructRecord}(\mathsf{pp_{TCM}}, \mathsf{apk}_j, \mathsf{payload}_j, \Phi_{\mathsf{b},j}, \Phi_{\mathsf{d},j}, \rho_j)$.
3. Retrieve current ledger digest: $\mathsf{st}_\mathbf{L} \leftarrow \mathbf{L}.\mathsf{Digest}$.
4. Construct instance $\mathbb{x}_\mathsf{e}$ for $\mathcal{R}_\mathsf{e}$: $\mathbb{x}_\mathsf{e} := (\mathsf{st}_\mathbf{L}, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo})$.
5. Construct witness $\mathbb{w}_\mathsf{e}$ for $\mathcal{R}_\mathsf{e}$: $\mathbb{w}_\mathsf{e} := ([\mathbf{r}_i]_1^m, [\mathbb{w}_{\mathbf{L},i}]_1^m, [\mathsf{ask}_i]_1^m, [\mathbf{r}_j]_1^n, \mathsf{aux})$.
6. Generate proof for $\mathcal{R}_\mathsf{e}$: $\pi_\mathsf{e} \leftarrow \mathsf{NIZK.Prove}(\mathsf{pp_e}, \mathbb{x}_\mathsf{e}, \mathbb{w}_\mathsf{e})$.
7. Construct transaction: $\mathsf{tx} := ([\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo}, \star)$, where $\star := (\mathsf{st}_\mathbf{L}, \pi_\mathsf{e})$.
8. Output $([\mathbf{r}_j]_1^n, \mathsf{tx})$.
```

```
$\mathsf{DPC.Verify}^\mathbf{L}$
Input: public parameters $\mathsf{pp}$ and transaction $\mathsf{tx}$
Output: decision bit $b$

1. Parse $\mathsf{tx}$ as $([\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo}, \star)$ and $\star$ as $(\mathsf{st}_\mathbf{L}, \pi_\mathsf{e})$.
2. Check that there are no duplicate serial numbers
   (a) within the transaction tx: $\mathsf{sn}_i \neq \mathsf{sn}_j$ for every distinct $i, j \in \{1, \ldots, m\}$;
   (b) on the ledger: $\mathbf{L}.\mathsf{Contains}(\mathsf{sn}_i) = 0$ for every $i \in \{1, \ldots, m\}$.
3. Check that the ledger state is valid: $\mathbf{L}.\mathsf{ValidateDigest}(\mathsf{st}_\mathbf{L}) = 1$.
4. Construct instance for the relation $\mathcal{R}_\mathsf{e}$: $\mathbb{x}_\mathsf{e} := (\mathsf{st}_\mathbf{L}, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo})$.
5. Check proof for the relation $\mathcal{R}_\mathsf{e}$: $\mathsf{NIZK.Verify}(\mathsf{pp_e}, \mathbb{x}_\mathsf{e}, \pi_\mathsf{e}) = 1$.
```

Figure 8: Construction of a DPC scheme.

$$\mathbb{x}_{\mathsf{e}} = \begin{pmatrix} \text{ledger digest} & \mathsf{st}_{\mathbf{L}} \\ \text{old record serial numbers} & [\mathsf{sn}_i]_1^m \\ \text{new record commitments} & [\mathsf{cm}_j]_1^n \\ \text{transaction memorandum} & \mathsf{memo} \end{pmatrix} \quad \text{and} \quad \mathbb{w}_{\mathsf{e}} = \begin{pmatrix} \text{old records} & [\mathbf{r}_i]_1^m \\ \text{old record membership witnesses} & [\mathbb{w}_{\mathbf{L},i}]_1^m \\ \text{old address secret keys} & [\mathsf{ask}_i]_1^m \\ \text{new records} & [\mathbf{r}_j]_1^n \\ \text{auxiliary predicate input} & \mathsf{aux} \end{pmatrix}$$

where

- for each $i \in \{1, \ldots, m\}$, $\mathbf{r}_i = (\mathsf{apk}_i, \mathsf{payload}_i, \Phi_{\mathsf{b},i}, \Phi_{\mathsf{d},i}, \rho_i, r_i, \mathsf{cm}_i)$;
- for each $j \in \{1, \ldots, n\}$, $\mathbf{r}_j = (\mathsf{apk}_j, \mathsf{payload}_j, \Phi_{\mathsf{b},j}, \Phi_{\mathsf{d},j}, \rho_j, r_j, \mathsf{cm}_j)$.

Define the local data $\mathsf{ldata} := \begin{pmatrix} [\mathsf{cm}_i]_1^m & [\mathsf{apk}_i]_1^m & [\mathsf{payload}_i]_1^m & [\Phi_{\mathsf{d},i}]_1^m & [\Phi_{\mathsf{b},i}]_1^m & [\mathsf{meta}_i]_1^m & [\mathsf{sn}_i]_1^m \\ [\mathsf{cm}_j]_1^n & [\mathsf{apk}_j]_1^n & [\mathsf{payload}_j]_1^n & [\Phi_{\mathsf{d},j}]_1^n & [\Phi_{\mathsf{b},j}]_1^n & \mathsf{memo} & \mathsf{aux} \end{pmatrix}$.

Then, a witness $\mathbb{w}_{\mathsf{e}}$ is valid for an instance $\mathbb{x}_{\mathsf{e}}$ if the following conditions hold:

1. For each $i \in \{i, \ldots, m\}$:
   - If $\mathbf{r}_i$ is not dummy, $\mathbb{w}_{\mathbf{L},i}$ proves that the commitment cm_i is in a ledger with digest $\mathsf{st}_{\mathbf{L}}$: $\mathbf{L}.\mathsf{Verify}(\mathsf{st}_{\mathbf{L}}, \mathsf{cm}_i, \mathbb{w}_{\mathbf{L},i}) = 1$.
   - The address public key apk_i and secret key ask_i form a valid key pair:
     $\mathsf{apk}_i = \mathsf{TCM.Commit}(\mathsf{pp_{TCM}}, \mathsf{sk}_{\mathsf{PRF},i} \| \mathsf{meta}_i;\ r_{\mathsf{pk},i})$ and $\mathsf{ask}_i = (\mathsf{sk}_{\mathsf{PRF},i}, \mathsf{meta}_i, r_{\mathsf{pk},i})$.
   - The serial number sn_i is valid: $\mathsf{sn}_i = \mathsf{PRF}_{\mathsf{sk}_{\mathsf{PRF},i}}(\rho_i)$.
   - The old record commitment cm_i is valid: $\mathsf{cm}_i = \mathsf{TCM.Commit}(\mathsf{pp_{TCM}}, \mathsf{apk}_i \| \mathsf{payload}_i \| \Phi_{\mathsf{b},i} \| \Phi_{\mathsf{d},i} \| \rho_i;\ r_i)$.
   - The death predicate $\Phi_{\mathsf{d},i}$ is satisfied by local data: $\Phi_{\mathsf{d},i}(i \| \mathsf{ldata}) = 1$.
2. For each $j \in \{1, \ldots, n\}$:
   - The serial number nonce ρ_j is computed correctly: $\rho_j = \mathsf{CRH.Eval}(\mathsf{pp_{CRH}}, j \| \mathsf{sn}_1 \| \ldots \| \mathsf{sn}_m)$.
   - The new record commitment cm_j is valid: $\mathsf{cm}_j = \mathsf{TCM.Commit}(\mathsf{pp_{TCM}}, \mathsf{apk}_j \| \mathsf{payload}_j \| \Phi_{\mathsf{b},j} \| \Phi_{\mathsf{d},j} \| \rho_j;\ r_j)$.
   - The birth predicate $\Phi_{\mathsf{b},j}$ is satisfied by local data: $\Phi_{\mathsf{b},j}(j \| \mathsf{ldata}) = 1$.

Figure 9: The execute NP relation $\mathcal{R}_{\mathsf{e}}$.

5 Delegating zero knowledge execution

The cost of creating a transaction in the DPC scheme from Section 4 grows with the complexity (and number of) predicates involved in the transaction. The user must produce, and include in the transaction, a cryptographic proof that, among other things, attests that death predicates of consumed records are satisfied and, similarly, that birth predicates of created records are satisfied. This implies that producing transactions on weak devices such as mobile phones or hardware tokens quickly becomes infeasible.

In Sections 5.1 to 5.3 we explain how to address this problem by enabling a user to *delegate* to an untrusted worker, such as a remote server, the computation that produces a transaction. This empowers weak devices to produce transactions that they otherwise could not have produced on their own. Then, in Section 5.4, we explain how the ideas that we use for delegating transactions also yield solutions for achieving *threshold transactions* and *blind transactions* in a DPC scheme, which are also valuable in applications. Techniques derived from these ideas are now part of deployed systems [HBHW18].

5.1 Approach

A naive approach is for the user to simply ask the worker to produce the cryptographic proof on its behalf, and then include this proof in the transaction. The intuition behind this idea is that the user can check that the proof received from the worker is valid, by simply running the proof verification procedure. Indeed, whenever the DPC scheme uses a succinct argument (see Remark 4.1), the verification procedure is *succinct*.

However, this approach is *insecure*, because the worker, in order to produce a proof, would have to learn not only the instance but also the secret witness for the NP statement being proved. Since the secret witness includes the user's address secret key, if the worker learns this information then the worker can impersonate the user, e.g., by producing further transactions that the user never *authorized*. This naive approach also fails in prior proof-based ledger protocols, including Zerocash [BCG$^+$14]. New ideas are needed.

Taking our construction of a DPC scheme from Section 4 as a starting point, we explain how to enable a user to delegate the expensive proof computation to a worker in such a way that the worker *cannot* produce valid transactions that have not been authorized by the user; see Fig. 11. (Additional security goals, such as ensuring that the worker learns no information about the user, are left to future work.)

The basic idea is to augment address keys in such a way that the secret information needed to produce the cryptographic proof is separate from the secret information needed to authorize a transaction containing that proof. Thus, the user can communicate to the worker the secrets necessary to generate a cryptographic proof, while retaining the remaining secrets for authorizing this (and future) transactions. In particular, the worker has no way to produce valid transactions that have not been authorized by the user.

We stress that the simplistic solution in which the user authorizes the proof produced by the worker by signing it via a secret key not shared with the worker *does not work* because it violates privacy. Indeed, others would have to use the same public key to verify signatures across multiple transactions containing signatures produced by the same secret key, thereby linking these transactions together.

The next two sub-sections explain how we achieve delegation: first, in Section 5.2, we describe a variant of randomizable signatures, which we use as a building block; then, in Section 5.3, we provide a high-level description of a *delegable* DPC scheme. The detailed construction is provided in Appendix B.

5.2 Additional building block: randomizable signatures

A *randomizable* signature scheme is a tuple of algorithms $\mathsf{SIG} = (\mathsf{Setup}, \mathsf{Keygen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{RandPk}, \mathsf{RandSig})$ that enables a party to sign messages, while also allowing randomization of public keys and signatures to prevent linking across multiple signatures. We first discuss the syntax of the usual algorithms.

- *Setup:* on input a security parameter, $\mathsf{SIG.Setup}$ samples public parameters $\mathsf{pp_{SIG}}$.
- *Key generation:* on input public parameters $\mathsf{pp_{SIG}}$, $\mathsf{SIG.Keygen}$ samples a key pair $(\mathsf{pk_{SIG}}, \mathsf{sk_{SIG}})$.
- *Message signing:* on input public parameters $\mathsf{pp_{SIG}}$, secret key $\mathsf{sk_{SIG}}$, and message m, $\mathsf{SIG.Sign}$ produces a signature σ.
- *Signature verification:* on input public parameters $\mathsf{pp_{SIG}}$, public key $\mathsf{pk_{SIG}}$, message m, and signature σ, $\mathsf{SIG.Verify}$ outputs a bit b denoting whether σ is a valid signature for m under public key $\mathsf{pk_{SIG}}$.

In addition to the usual algorithms, SIG has two algorithms for randomizing public keys and signatures.

- *Public key randomization:* $\mathsf{SIG.RandPk}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}}, r_{\mathsf{SIG}})$ samples a randomized public key $\hat{\mathsf{pk}}_{\mathsf{SIG}}$.
- *Signature randomization:* $\mathsf{SIG.RandSig}(\mathsf{pp_{SIG}}, \sigma, r_{\mathsf{SIG}})$ samples a randomized signature $\hat{\sigma}$.

The signature scheme SIG must satisfy the following security properties.

- *Existential unforgeability.* Given a public key $\mathsf{pk_{SIG}}$, it is infeasible to produce a forgery under $\mathsf{pk_{SIG}}$ or *under under any randomization of* $\mathsf{pk_{SIG}}$. This notion strengthens the standard unforgeability notion, and is similar to that of randomizable signatures in [FKM$^+$16].
- *Unlinkability.* Given a public key $\mathsf{pk_{SIG}}$ and a tuple $(\hat{\mathsf{pk}}_{\mathsf{SIG}}, m, \hat{\sigma})$ where $\hat{\sigma}$ is a valid signature for m under $\hat{\mathsf{pk}}_{\mathsf{SIG}}$, no efficient adversary can determine if $\hat{\mathsf{pk}}_{\mathsf{SIG}}$ is a fresh public key and $\hat{\sigma}$ a fresh signature, or if instead $\hat{\mathsf{pk}}_{\mathsf{SIG}}$ is a randomization of $\mathsf{pk_{SIG}}$ and $\hat{\sigma}$ a randomization of a signature for $\mathsf{pk_{SIG}}$. This property is a computational relaxation of the perfect unlinkability property of randomizable signatures in [FKM$^+$16].
- *Injective randomization.* Randomization of public keys is *(computationally) injective* with respect to randomness. Informally, given public parameters $\mathsf{pp_{SIG}}$, it is infeasible to find a public key $\mathsf{pk_{SIG}}$ and $r_1 \neq r_2$ such that $\mathsf{SIG.RandPk}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}}, r_1) = \mathsf{SIG.RandPk}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}}, r_2)$.

5.3 A delegable DPC scheme

We describe how to construct a *delegable DPC scheme*, namely, a DPC scheme in which a user can delegate to an untrusted worker the expensive computations associated with producing a transaction. The security goal is that the worker should not be able to produce valid transactions that have not been authorized by the user. Below we assume familiarity with our "plain" DPC construction (see Section 4).

The user will maintain (among other things) a key pair $(\mathsf{pk_{SIG}}, \mathsf{sk_{SIG}})$ for a randomizable signature scheme SIG (see Section 5.2). The public key $\mathsf{pk_{SIG}}$ will be embedded in the user's public key apk and also be used to derive the serial numbers of records "owned" by apk. In contrast, the secret key $\mathsf{sk_{SIG}}$ will not be a part of any data structures, and will *only* be used to *authorize* transactions by signing the cryptographic proofs produced by untrusted workers.

In more detail, we first describe how addresses and records are generated (also see summary in Fig. 10).

- **Addresses.** In Section 4 an address public key apk was a trapdoor commitment to a secret key $\mathsf{sk_{PRF}}$ for a pseudorandom function PRF and the address metadata meta. Now apk is a trapdoor commitment to this same information *as well as* the public key of a key pair $(\mathsf{pk_{SIG}}, \mathsf{sk_{SIG}})$ for SIG. The corresponding address secret key ask consists of all the committed information and the commitment randomness.

- **Records.** The structure of a record, including how a record commitment is computed, is as in Section 4. However, a record's serial number sn is now derived in a different way: while previously $\mathsf{sn} := \mathsf{PRF}_{\mathsf{sk_{PRF}}}(\rho)$ now we set $\mathsf{sn} := \mathsf{SIG.RandPk}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}}, \mathsf{PRF}_{\mathsf{sk_{PRF}}}(\rho))$ where ρ is the record's serial number nonce. Namely, while before serial numbers were outputs of a pseudorandom function keyed by $\mathsf{sk_{PRF}}$, now they are randomizations of the authorization public key $\mathsf{pk_{SIG}}$ when using suitable pseudorandomness.

Note that the foregoing new derivation of serial numbers does not break important security properties.

- *Unlinkability of serial numbers:* serial numbers of different records that share the same authorization public $\mathsf{pk}_{\mathsf{SIG}}$ are computationally indistinguishable. This follows rather directly from the fact sn, being a randomization of $\mathsf{pk}_{\mathsf{SIG}}$, does not reveal information (to efficient distinguishers) about $\mathsf{pk}_{\mathsf{SIG}}$ itself.
- *No double spending:* a user cannot "spend" (i.e., consume) $\mathbf{r}$ in two different transactions by revealing different serial numbers because r_{SIG} (and thus sn) is generated deterministically from $\mathbf{r}$. Since SIG is randomness-injective in $\mathsf{SIG.RandPk}$, sn is (computationally) unique to $\mathbf{r}$.

Having described the modified data structures of addresses and serial numbers, we now explain how a user can task a worker to produce the cryptographic proofs that need to be included in a transaction. For simplicity, in this high-level discussion we focus on the case where the transaction involves only one input (old) record $\mathbf{r}$ and one output (new) record $\mathbf{r}'$. In this case, the transaction contains a serial number sn (supposedly corresponding to $\mathbf{r}$), and a commitment cm' (supposedly corresponding to $\mathbf{r}'$).

Previously, the user had to generate a proof π_{e} that sn is consistent with $\mathbf{r}$, that cm' can be opened to $\mathbf{r}'$, and that the death and birth predicates of $\mathbf{r}$ and $\mathbf{r}'$ respectively are satisfied. Now the user can delegate to a worker the generation of the proof π_{e} because the modified derivation of apk and sn allows the user to communicate to the worker only $\mathbf{r}$, $\mathbf{r}'$ and a *part* of the address secret key of $\mathbf{r}$. Namely, the user sends to the worker only the pseudorandom function key $\mathsf{sk}_{\mathsf{PRF}}$ and the commitment randomness r_{pk}. Crucially, the user does not have to communicate to the worker the authorization secret key $\mathsf{sk}_{\mathsf{SIG}}$.

After receiving the proof π_{e} from the worker, the user uses the authorization secret key $\mathsf{sk}_{\mathsf{SIG}}$ to sign π_{e} (along with the instance that π_{e} attests to), and then randomizes the resulting signature σ to obtain $\hat{\sigma}$. The final transaction tx not only includes the serial number sn (consuming the old record), the commitment cm' (creating the new record), and π_{e} (attesting to the correct state transition) as before, but also includes $\hat{\sigma}$. Transaction verification involves checking the proof π_{e} and also checking that $\hat{\sigma}$ is valid with respect to the randomized public key sn.

This completes our high-level description of our delegable DPC scheme; see Appendix B for details.

	Plain DPC	Delegable DPC
Address secret key	$(\mathsf{sk}_{\mathsf{PRF}}, \mathsf{meta}, r_{\mathsf{pk}})$	$(\mathsf{sk}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{PRF}}, \mathsf{meta}, r_{\mathsf{pk}})$
Address public key	$\mathsf{apk} := \mathsf{TCM.Commit}\left(\begin{matrix}\mathsf{pp}_{\mathsf{TCM}}, \\ \mathsf{sk}_{\mathsf{PRF}} \Vert \mathsf{meta}\end{matrix} ; r_{\mathsf{pk}}\right)$	$\mathsf{apk} := \mathsf{TCM.Commit}\left(\begin{matrix}\mathsf{pp}_{\mathsf{TCM}}, \\ \mathsf{pk}_{\mathsf{SIG}} \Vert \mathsf{sk}_{\mathsf{PRF}} \Vert \mathsf{meta}\end{matrix} ; r_{\mathsf{pk}}\right)$
Serial number derivation	$\mathsf{sn} \leftarrow \mathsf{PRF}_{\mathsf{sk}_{\mathsf{PRF}}}(\rho)$	1. $r_{\mathsf{SIG}} \leftarrow \mathsf{PRF}_{\mathsf{sk}_{\mathsf{PRF}}}(\rho)$ 2. $\mathsf{sn} \leftarrow \mathsf{SIG.RandPk}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{SIG}}, r_{\mathsf{SIG}})$
Transaction construction	$\mathsf{tx} := ([\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo}, \star)$, where $\star := (\mathsf{st}_{\mathbf{L}}, \pi_{\mathsf{e}})$.	1. **Sign transaction contents:** (a) $\sigma_i \leftarrow \mathsf{SIG.Sign}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{SIG},i}, \mathbb{x}_{\mathsf{e}} \Vert \pi_{\mathsf{e}})$. (b) $\hat{\sigma}_i \leftarrow \mathsf{SIG.RandSig}(\mathsf{pp}_{\mathsf{SIG}}, \sigma_i, r_{\mathsf{SIG},i})$. 2. $\mathsf{tx} := ([\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo}, \star)$, where $\star := (\mathsf{st}_{\mathbf{L}}, \pi_{\mathsf{e}}, [\hat{\sigma}_i]_1^m)$.
Transaction verification	Check that serial numbers do not appear on ledger, that the ledger state digest is valid, and that the NIZK proof verifies.	As in plain DPC, but additionally check that each **signature verifies:** $\mathsf{SIG.Verify}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{tx.sn}_i, \mathbb{x}_{\mathsf{e}} \Vert \pi_{\mathsf{e}}, \sigma_i) = 1$.

Figure 10: Summary of differences between plain DPC and delegable DPC (highlighted).

5.4 Threshold transactions and blind transactions

We explain how the delegable DPC scheme described above can be modified, in a straightforward way, to achieve additional features: *threshold transactions* or *blind transactions*.

Threshold transactions. A DPC scheme has threshold transactions if the power to authorize transactions can be vested unto any t out of n parties, for any desired choice of t and n (as opposed to a single user as discussed thus far, which corresponds to the special case of $t = n = 1$); see Fig. 12. Threshold transactions are useful in many settings, e.g., to enhance operational security by realizing two-factor authentication.

We can achieve threshold transactions by simply using, in our delegable DPC scheme, a randomizable signature scheme SIG that also supports *threshold key generation* and *threshold signing algorithms* [DF91]. Such a *threshold signature scheme* distributes signing ability among n parties such that at least t of them are needed to authorize a signature. Threshold key generation would then be used to create an address, and threshold signing would be used to authorize a transaction by signing the corresponding cryptographic proof.

Blind transactions. A DPC scheme has blind transactions if there is a way for a user to authorize a transaction without learning of its contents; see Fig. 13. Blind transactions, in conjunction with prior techniques [CGL$^+$17], can be used to construct efficient lottery tickets and thereby probabilistic micropayments.

We can achieve blind transactions by simply using, in our delegable DPC scheme, a randomizable signature scheme SIG that has a *blind signing algorithm*, which can then be used for signing the relevant cryptographic proof in order to authorize a transaction.

Instantiating randomizable threshold and blind signatures. As we explain in Appendix B.1, we construct randomizable signature schemes by modifying Schnorr signatures. To further construct threshold or blind randomizable signatures, it is enough to note that public key and signature randomization occurs *after* the public key or signature has been created. Thus one can use existing protocols for threshold key-generation and signing [SS01, NKDM03, Dod07], and blind signing [PS00, SJ99] to obtain public keys and signatures, and then use the algorithms from Appendix B.1 to randomize these. A nice feature of this approach is that all these types of delegated transactions (regular, threshold, blind) cannot be distinguished from one another.

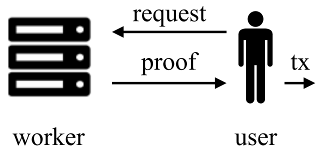

Figure 11: Delegable transactions.

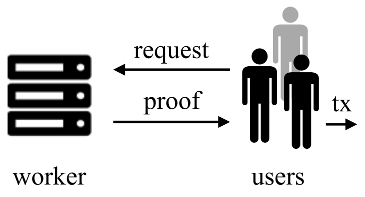

Figure 12: Threshold transactions.

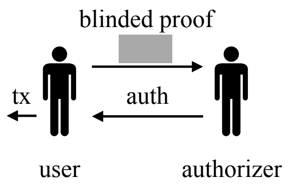

Figure 13: Blind transactions.

6 A simple example: user-defined assets

In this section we provide several example applications of Zexe, and demonstrate how these can be "programmed" within the records nano-kernel. In this section we draw inspiration from current applications for smart contract systems such as Ethereum, which are largely focused on financial applications. We note that in these settings, privacy of contract inputs can be particularly valuable, given the monetary advantage that information asymmetry provides traders.

Payments and user-defined assets. One of the most basic applications of smart contract systems like Ethereum is the construction of assets (or "tokens") that can be used for financial applications. For example, the Ethereum ERC20 specification [VB15] defines a general framework for such assets. These assets have two phases: asset minting (creation), and asset conservation (expenditure).

In more detail, the birth predicate $\Phi_{\mathsf{b}}^{\star}$ can be invoked in two modes, *mint mode* or *conserve mode*. When invoked in mint mode, $\Phi_{\mathsf{b}}^{\star}$ creates the initial supply v of the asset in, say, a single output record, by deterministically deriving a fresh unique identifier id for the asset,[8] and storing the pair (id, v) in the record's payload. The predicate $\Phi_{\mathsf{b}}^{\star}$ also ensures that in the given transaction there are no input records or other output records (dummy records are allowed). If $\Phi_{\mathsf{b}}^{\star}$ is invoked in mint mode in other transactions, a *different* identifier id is created, ensuring that multiple assets can be reliably distinguished even though anyone can run $\Phi_{\mathsf{b}}^{\star}$.

When invoked in conserve mode, $\Phi_{\mathsf{b}}^{\star}$ inspects all records in a transaction whose birth predicates all equal $\Phi_{\mathsf{b}}^{\star}$ (i.e., all the transaction's user-defined assets) and whose asset identifiers all equal to the identifier of the current record. For these records it ensures that the no new value is created: that is, the sum of the value across all output records is less than or equal to the sum of the value in all input records.

Custom access to assets. We can further refine the above application by noting that users can program death predicates to dictate access controls to their funds. This akin to how in Bitcoin scripts specify conditions on how funds can be spent (except that in Bitcoin scripts have limited expressivity). Access control mechanisms could vary from saying that the transaction must be authorized by two of three public keys, is valid only after a given amount of time, or must reveal the pre-image of a hash function. For example, this can be used to create so-called hash-timelock transactions that enable payment channels or enforce that a transaction must be authorized by multiple parties. We work out a concrete example next.

Conditional exchange of assets. In this application we want to enable users to exchange user-defined assets. For example, suppose that Alice owns 100 units of an asset with identifier id_1, and wishes to exchange them for 50 units of an asset with identifier id_2, but does not have a counter-party for the exchange. She could create a record $\mathbf{r}$ containing 100 units of id_1 that is consumable (i.e. payable to) anyone provided that it is done in a transaction that pays Alice 50 units of id_2. She can do so by simply setting the death predicate to require that the transaction consuming $\mathbf{r}$ also creates a record $\mathbf{r}'$ belonging to Alice for 50 units of asset id_2. Alice then publishes a transaction creating $\mathbf{r}$ and out-of-band announces the offer along with the details needed to claim it (including the contents of the record and the secret of of the address owning it). Anyone with this knowledge can take on Alice's offer by creating a transaction that: (a) consumes the 100 units of id_1 in $\mathbf{r}$ from Alice; (b) consumes 50 units of id_2 in some other records; (c) creates 50 units of id_2 in $\mathbf{r}'$ for Alice; (d) creates 100 units of id_1 in some other records.

[8] One can securely generate this unique identifier by setting it to be the serial number of a dummy record that is consumed during the creation of the initial supply; since serial numbers are globally unique, so is the identifier.

7 Implementation strategy

The straightforward approach to implement our construction of a DPC scheme (described in Section 4) is to instantiate the proof system via a simulation-extractable zkSNARK (e.g., [GM17]) and then select the other cryptographic building blocks so that the circuit (more precisely, constraint system) for deciding the NP relation $\mathcal{R}_{\mathsf{e}}$ has as small a size as possible. While the straightforward approach sounds promising, closer inspection reveals significant costs that we need to somehow reduce. In this section we discuss, in a "problem and solution" format, the challenges that we encountered and how we addressed them. (The implementation strategy for plain DPC schemes directly ports over to delegable DPC schemes so we do not discuss them.)

Problem 1: universality is expensive. The NP relation $\mathcal{R}_{\mathsf{e}}$ involves checking arbitrary predicates, which means that one must rely on proof systems for *universal* computations. However, checking universal computations via state-of-the-art zkSNARKs involves expensive tools for universal circuits/machines [BCG$^+$13, BCTV14, WSR$^+$15, BCTV17]. These tools would not only yield an expensive solution but would also penalize users who only produce transactions that attest to simple inexpensive predicates, because these users would have to incur the costs of using these "heavy duty" proof systems.

Solution 1: recursive proof verification. We address this problem by relying on one layer of *recursive proof composition* [Val08, BCCT13]. Instead of tasking $\mathcal{R}_{\mathsf{e}}$ with checking satisfiability of general predicates, we only task it with checking *succinct proofs* attesting to this. Checking succinct proofs is a (relatively) inexpensive computation that is universal for *all* predicates, which can be "hardcoded" in $\mathcal{R}_{\mathsf{e}}$. Crucially, since the "outer" succinct proofs produced for $\mathcal{R}_{\mathsf{e}}$ do not reveal information about the "inner" succinct proofs attesting to predicates' satisfiability (thanks to zero knowledge), the inner succinct proofs do *not* have to hide what predicate was checked, removing the need for expensive universal circuits; in fact, inner proofs do not even have to be zero knowledge. Rather, these inner succinct proofs can be for NP relations *tailored* to the computations needed by particular birth and death predicates. Furthermore, this approach ensures that a user only has to incur the cost of proving satisfiability of the specific predicates involved in his transactions, regardless of the complexity of predicates used by other users in their transactions.

In more detail, taking the case of one input and one output record as an example, we modify $\mathsf{DPC.Execute}$ to additionally take as input SNARK proofs π_{d} and π_{b}, and also modify the NP relation $\mathcal{R}_{\mathsf{e}}$ so that, instead of directly checking that Φ_{d} and Φ_{b} are satisfied, it instead checks that π_{d} *and* π_{b} *attest to the satisfaction of* Φ_{d} *and* Φ_{b}. That is, $\mathcal{R}_{\mathsf{e}}$ checks that $\mathsf{NIZK.Verify}(\mathsf{pp}_{\Phi_{\mathsf{d}}}, \mathbb{x}_{\mathsf{e}}, \pi_{\mathsf{d}}) = 1$ and $\mathsf{NIZK.Verify}(\mathsf{pp}_{\Phi_{\mathsf{b}}}, \mathbb{x}_{\mathsf{e}}, \pi_{\mathsf{b}}) = 1$, where $\mathsf{pp}_{\Phi_{\mathsf{d}}}$ are public parameters for the NP relation $\mathcal{R}_{\Phi_{\mathsf{d}}} := \{(\mathbb{x}_{\mathsf{e}}, \mathbb{w}_{\mathsf{e}}) \text{ s.t. } \Phi_{\mathsf{d}}(\mathbb{x}_{\mathsf{e}}, \mathbb{w}_{\mathsf{e}}) = 1\}$ and similarly for Φ_{b}. The public parameters $\mathsf{pp}_{\Phi_{\mathsf{d}}}$ and $\mathsf{pp}_{\Phi_{\mathsf{b}}}$ are stored in the record, in place of (a description of) the predicates.[9]

More generally, we modify $\mathsf{DPC.Execute}$ to additionally take as input SNARK proofs $[\pi_{\mathsf{d},i}]_1^m$ attesting that the old records' death predicates are satisfied and SNARK proofs $[\pi_{\mathsf{b},j}]_1^n$ attesting that the new records' birth predicates are satisfied. Moreover, we similarly modify the NP relation $\mathcal{R}_{\mathsf{e}}$ to check that these proofs are valid, instead of directly checking that the relevant predicates are satisfied.

In sum, $\mathcal{R}_{\mathsf{e}}$ is not tasked with checking general predicates. Instead, it merely has to check SNARK proofs, a fixed computation of size $O_\lambda(m+n)$. Separately, a user wishing to prove that a predicate Φ is satisfied will invoke a SNARK on an NP statement of size $|\Phi|$ (tailored for Φ).[10] The approach described so far, however, hides additional costs that we need to overcome.

[9]More precisely, to verify a proof for a predicate Φ, the proof verifier does not need to read all of pp_Φ, which has size $O_\lambda(|\Phi|)$ in some zkSNARKs (i.e., it is large). Rather, the proof verifier only needs to read $O_\lambda(|\mathbb{x}_{\mathsf{e}}|)$ bits of pp_Φ, which are collectively known as the *verification key*. The record would then store this verification key (or a hash thereof) rather than pp_Φ.

[10]An additional benefit of each predicate Φ having its own public parameters pp_Φ is flexible trust: users are not obliged to trust parameters used in each others' transactions and, moreover, if some parameters are known to be compromised, predicates can safely refuse to interact with records associated with them. We view this *isolation mechanism* as a novel and valuable feature in practice.

Problem 2: recursion is expensive. Recursive proof composition has so far been empirically demonstrated for pairing-based SNARKs [BCTV17], whose proofs are extremely short and cheap to verify. We thus focus our attention on these, and explain the efficiency challenges that we must overcome in our setting.

Recall that pairings are instantiated via elliptic curves of small embedding degree. If we instantiate a SNARK's pairing via an elliptic curve E defined over a prime field $\mathbb{F}_q$ and having a subgroup of large prime order r, then (a) the SNARK supports NP relations $\mathcal{R}$ expressed as arithmetic circuits over $\mathbb{F}_r$, while (b) proof verification involves arithmetic operations over $\mathbb{F}_q$. This means that we need to express $\mathcal{R}_{\mathsf{e}}$ via arithmetic circuits over $\mathbb{F}_r$. In turn, since the SNARK verifier is part of $\mathcal{R}_{\mathsf{e}}$, this means that we need to also express the verifier via an arithmetic circuit over $\mathbb{F}_r$, which is problematic because the verifier's "native" operations are over $\mathbb{F}_q$. Simulating $\mathbb{F}_q$ operations via $\mathbb{F}_r$ operations introduces significant overheads, and picking E such that $q = r$, in order to avoid simulation, is impossible [BCTV17].

Prior work thus suggests using *multiple* curves [BCTV17], such as a two-cycle of pairing-friendly elliptic curves, that is, two prime-order curves E_1 and E_2 such that the prime size of one's base field is the prime order of the other's group, and orchestrating SNARKs based on these so that fields always "match up". Unfortunately, known curves with these properties are inefficient at 128 bits of security [BCTV17, CCW18].

Solution 2: tailored set of curves. In our setting we merely need "a proof of a proof", with the latter proof not itself depending on further proofs. This implies that we do not actually need a cycle of pairing-friendly elliptic curves (which enables recursion of arbitrary depth) and, in particular, we can use the Cocks–Pinch method [FST10] to set up a bounded recursion [BCTV17]. We now elaborate on this.

First we pick a pairing-friendly elliptic curve that not only is suitable for 128 bits of security according to standard considerations (involving, e.g., its embedding degree and the ratio of the sizes of its base field and prime order group) but, moreover, is compatible with efficient SNARK provers in *both* levels of the recursion. Namely, letting p be the prime order of the base field and r the prime order of the group, we need that *both* $\mathbb{F}_r$ and $\mathbb{F}_p$ have multiplicative subgroups whose orders are large powers of 2. The condition on $\mathbb{F}_r$ ensures efficient proving for SNARKs over this curve, while the condition on $\mathbb{F}_p$ ensures efficient proving for SNARKs that verify proofs over this curve. In light of the above, we select a curve E_{BLS} from the Barreto–Lynn–Scott (BLS) family [BLS02, CLN11] with embedding degree 12. This family not only enables parameters that conservatively achieve 128 bits of security, but also enjoys properties that facilitate very efficient implementation [AFK$^+$12]. We ensure that both $\mathbb{F}_r$ and $\mathbb{F}_p$ have multiplicative subgroups of order 2^α for $\alpha \geq 40$, by choosing the parameter x of the BLS family to satisfy $x \equiv 1 \bmod 3 \cdot 2^\alpha$; indeed, for such a choice of x both $r(x) = x^4 - x^2 + 1$ and $p(x) = (x-1)^2 r(x)/3 + x$ are divisible by 2^α. This also ensures that $x \equiv 1 \bmod 3$, which ensures that there are efficient towering options for the relevant fields [Cos12].

Next we use the Cocks–Pinch method to pick a pairing-friendly elliptic curve E_{CP} over a field $\mathbb{F}_q$ such that the curve group $E_{\mathsf{CP}}(\mathbb{F}_q)$ contains a subgroup of prime order p (the size of E_{BLS}'s base field). Since the method outputs a prime q that has about $2\times$ more bits than the desired p, and in turn p has about $1.5\times$ more bits than r (due to properties of the BLS family), we only need E_{CP} to have embedding degree 6 in order to achieve 128 bits of security (as determined from the guidelines in [FST10]).

In sum, proofs of predicates' satisfiability are produced via a SNARK over E_{BLS}, and proofs for the NP relation $\mathcal{R}_{\mathsf{e}}$ are produced via a zkSNARK over E_{CP}. The matching fields between the two curves ensure that the former proofs can be efficiently verified.

Problem 3: Cocks–Pinch curves are costly. While the curve E_{CP} was chosen to facilitate efficient checking of proofs over E_{BLS}, the curve E_{CP} is at least $2\times$ more expensive (in time and space) than E_{BLS} simply because E_{CP}'s base field has about twice as many bits as E_{BLS}'s base field. Checks in the NP relation $\mathcal{R}_{\mathsf{e}}$ that are not directly related to proof checking are now unnecessarily carried over a less efficient curve.

Solution 3: split relations across two curves. We split $\mathcal{R}_{\mathsf{e}}$ into two NP relations $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$ (see

Fig. 14), with the latter containing just the proof check and the former containing all other checks. We can then use a zkSNARK over the curve E_{BLS} (an efficient curve) to produce proofs for $\mathcal{R}_{\mathsf{BLS}}$, and a zkSNARK over E_{CP} (the less efficient curve) to produce proofs for $\mathcal{R}_{\mathsf{CP}}$. This approach significantly reduces the running time of DPC.Execute (producing proofs for the checks in $\mathcal{R}_{\mathsf{BLS}}$ is more efficient over E_{BLS} than over E_{CP}), at the expense of a modest increase in transaction size (a transaction now includes a zkSNARK proof over E_{BLS} in addition to a proof over E_{CP}). An important technicality that must be addressed is that the foregoing split relies on certain secret information to be shared across the NP relations, namely, the identities of relevant predicates and the local data. We can store this information in suitable commitments that are part of the NP instances for the two NP relations (doing this efficiently requires some care as we discuss below).

Problem 4: the NP relations have many checks. Even if we moved all checks not tied to the curve E_{CP} to the more efficient curve E_{BLS}, the NP relation $\mathcal{R}_{\mathsf{e}}$ (more precisely, its split into $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$) collectively involves many checks that range from verifying authentication paths in a Merkle tree to verifying commitment openings, and from evaluating pseudorandom functions to evaluating collision resistant functions. NP relations with similar structure, such as those used in Zerocash [BCG$^+$14], required upwards of *four million gates* to express all of these checks. This not only resulted in high latencies for producing transactions (several minutes) but also resulted in large public parameters for the system (hundreds of megabytes).

Solution 4: efficient EC primitives. Commitments and collision-resistant hashing can be expressed as very efficient arithmetic circuits if one opts for Pedersen-type constructions over suitable Edwards elliptic curves (and techniques derived from these ideas are now part of deployed systems [HBHW18]). To do this, we pick two Edwards curves, $E_{\mathsf{Ed/BLS}}$ over the field $\mathbb{F}_r$ (matching the group order of E_{BLS}) and $E_{\mathsf{Ed/CP}}$ over the field $\mathbb{F}_p$ (matching the group order of E_{CP}). This enables us to achieve very efficient circuits for primitives used in our NP relations, including commitments, collision-resistant hashing, and randomizable signatures. (Note that $E_{\mathsf{Ed/BLS}}$ and $E_{\mathsf{Ed/CP}}$ do not need to be pairing-friendly as the primitives only rely on their group structure.)

Problem 5: sharing information between NP relations is costly. We have said that splitting $\mathcal{R}_{\mathsf{e}}$ into two NP relations $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$ relies on sharing secret information via commitments across NP statements; namely, a commitment cm_{Φ} to the identities of predicates and a commitment $\mathsf{cm}_{\mathsf{ldata}}$ to the local data. But if both relations open these commitments, we cannot make an efficient use of Pedersen commitments because the two NP relations are over different fields: $\mathcal{R}_{\mathsf{BLS}}$ is over $\mathbb{F}_r$, while $\mathcal{R}_{\mathsf{CP}}$ is over $\mathbb{F}_p$. For example, if we used a Pedersen commitment over the order-r subgroup of the Edwards curve $E_{\mathsf{Ed/BLS}}$, then: (a) opening a commitment in $\mathcal{R}_{\mathsf{BLS}}$ would be cheap, but (b) opening a commitment in $\mathcal{R}_{\mathsf{CP}}$ would involve expensive simulation of $\mathbb{F}_r$-arithmetic via $\mathbb{F}_p$-arithmetic. (And similarly if we used a Pedersen commitment over the order-p subgroup of the Edwards curve $E_{\mathsf{Ed/CP}}$.) To make matters worse, the predicate identities and the local data are large, so an inefficient solution for committing to these would add significant costs to $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$.

Solution 5: hash predicate verification keys and commit to local data. In a record, instead of storing predicate verification keys, we store collision-resistant hashes of these. This reduces the cost of producing the commitment cm_{Φ} in $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$, as cm_{Φ} contains hashes that are much smaller than verification keys. We realize cm_{Φ} via Blake2s, a boolean primitive of modest cost in $\mathbb{F}_r$ and $\mathbb{F}_p$. Crucially, *only* $\mathcal{R}_{\mathsf{CP}}$ needs to access the verification keys themselves, so we can efficiently use a Pedersen hash over the Edwards curve $E_{\mathsf{Ed/CP}}$ to let $\mathcal{R}_{\mathsf{CP}}$ check the keys (supplied as non-deterministic advice) against the hashes inside cm_{Φ}.

We realize the local data commitment $\mathsf{cm}_{\mathsf{ldata}}$ via a Pedersen commitment over $E_{\mathsf{Ed/BLS}}$, and assume that predicates take $\mathsf{cm}_{\mathsf{ldata}}$ as input rather than local data in the clear. Since both $\mathcal{R}_{\mathsf{BLS}}$ and the predicate relations are defined over the field $\mathbb{F}_r$ (the prime-order subgroup of the curve E_{BLS}), non-deterministically opening $\mathsf{cm}_{\mathsf{ldata}}$ is efficient in both relations. This approach significantly reduces costs because $\mathcal{R}_{\mathsf{CP}}$ no longer needs to reason about the contents of $\mathsf{cm}_{\mathsf{ldata}}$, and can simply pass $\mathsf{cm}_{\mathsf{ldata}}$ as input to the SNARK verifier.

The NP relation $\mathcal{R}_{\mathsf{BLS}}$ has instances $\mathbb{x}_{\mathsf{BLS}}$ and witnesses $\mathbb{w}_{\mathsf{BLS}}$ of the form

$$\mathbb{x}_{\mathsf{BLS}} = \begin{pmatrix} \text{ledger digest} & \mathsf{st}_{\mathbf{L}} \\ \text{old record serial numbers} & [\mathsf{sn}_i]_1^m \\ \text{new record commitments} & [\mathsf{cm}_j]_1^n \\ \text{predicate commitment} & \mathsf{cm}_\Phi \\ \text{local data commitment} & \mathsf{cm}_{\mathsf{ldata}} \\ \text{transaction memorandum} & \mathsf{memo} \end{pmatrix} \quad \text{and} \quad \mathbb{w}_{\mathsf{BLS}} = \begin{pmatrix} \text{old records} & [\mathbf{r}_i]_1^m \\ \text{old record membership witnesses} & [\mathbb{w}_{\mathbf{L},i}]_1^m \\ \text{old address secret keys} & [\mathsf{ask}_i]_1^m \\ \text{new records} & [\mathbf{r}_j]_1^n \\ \text{predicate comm. randomness} & r_\Phi \\ \text{local data randomness} & r_{\mathsf{ldata}} \\ \text{auxiliary predicate input} & \mathsf{aux} \end{pmatrix}$$

where

- for each $i \in \{1, \dots, m\}$, $\mathbf{r}_i = (\mathsf{apk}_i, \mathsf{payload}_i, h_{\mathsf{b},i}, h_{\mathsf{d},i}, \rho_i, r_i, \mathsf{cm}_i)$;
- for each $j \in \{1, \dots, n\}$, $\mathbf{r}_j = (\mathsf{apk}_j, \mathsf{payload}_j, h_{\mathsf{b},j}, h_{\mathsf{d},j}, \rho_j, r_j, \mathsf{cm}_j)$.

Define the local data $\mathsf{ldata} := \begin{pmatrix} [\mathsf{cm}_i]_1^m & [\mathsf{apk}_i]_1^m & [\mathsf{payload}_i]_1^m & [h_{\mathsf{d},i}]_1^m & [h_{\mathsf{b},i}]_1^m & [\mathsf{meta}_i]_1^m & [\mathsf{sn}_i]_1^m \\ [\mathsf{cm}_j]_1^n & [\mathsf{apk}_j]_1^n & [\mathsf{payload}_j]_1^n & [h_{\mathsf{d},j}]_1^n & [h_{\mathsf{b},j}]_1^n & \mathsf{memo} & \mathsf{aux} \end{pmatrix}$.

A witness $\mathbb{w}_{\mathsf{BLS}}$ is valid for an instance $\mathbb{x}_{\mathsf{BLS}}$ if the following conditions hold:

1. For each $i \in \{i, \dots, m\}$:
   - If $\mathbf{r}_i$ is not dummy, $\mathbb{w}_{\mathbf{L},i}$ proves that the commitment cm_i is in a ledger with digest $\mathsf{st}_{\mathbf{L}}$: $\mathbf{L}.\mathsf{Verify}(\mathsf{st}_{\mathbf{L}}, \mathsf{cm}_i, \mathbb{w}_{\mathbf{L},i}) = 1$.
   - The address public key apk_i and secret key ask_i form a valid key pair:
     $\mathsf{apk}_i = \mathsf{TCM.Commit}(\mathsf{pp}_{\mathsf{TCM}}, \mathsf{sk}_{\mathsf{PRF},i} \| \mathsf{meta}_i;\ r_{\mathsf{pk},i})$ and $\mathsf{ask}_i = (\mathsf{sk}_{\mathsf{PRF},i}, \mathsf{meta}_i, r_{\mathsf{pk},i})$.
   - The serial number sn_i is valid: $\mathsf{sn}_i = \mathsf{PRF}_{\mathsf{sk}_{\mathsf{PRF},i}}(\rho_i)$.
   - The old record commitment cm_i is valid: $\mathsf{cm}_i = \mathsf{TCM.Commit}(\mathsf{pp}_{\mathsf{TCM}}, \mathsf{apk}_i \| \mathsf{payload}_i \| h_{\mathsf{b},i} \| h_{\mathsf{d},i} \| \rho_i;\ r_i)$.
2. For each $j \in \{1, \dots, n\}$:
   - The serial number nonce ρ_j is computed correctly: $\rho_j = \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, j \| \mathsf{sn}_1 \| \dots \| \mathsf{sn}_m)$.
   - The new record commitment cm_j is valid: $\mathsf{cm}_j = \mathsf{TCM.Commit}(\mathsf{pp}_{\mathsf{TCM}}, \mathsf{apk}_j \| \mathsf{payload}_j \| h_{\mathsf{b},j} \| h_{\mathsf{d},j} \| \rho_j;\ r_j)$.
3. The predicate commitment cm_Φ is valid: $\mathsf{cm}_\Phi = \mathsf{b2s}([h_{\mathsf{d},i}]_1^m \ \|\ [h_{\mathsf{b},j}]_1^n \ \|\ r_\Phi)$.
4. The local data commitment $\mathsf{cm}_{\mathsf{ldata}}$ is valid: $\mathsf{cm}_{\mathsf{ldata}} = \mathsf{CM.Commit}(\mathsf{pp}_{\mathsf{CM}}, \mathsf{ldata}; r_{\mathsf{ldata}})$

The NP relation $\mathcal{R}_{\mathsf{CP}}$ has instances $\mathbb{x}_{\mathsf{CP}}$ and witnesses $\mathbb{w}_{\mathsf{CP}}$ of the form

$$\mathbb{x}_{\mathsf{CP}} = \begin{pmatrix} \text{predicate commitment} & \mathsf{cm}_\Phi \\ \text{local data commitment} & \mathsf{cm}_{\mathsf{ldata}} \end{pmatrix} \quad \text{and} \quad \mathbb{w}_{\mathsf{CP}} = \begin{pmatrix} \text{old death pred. ver. keys} & [\mathsf{vk}_{\mathsf{d},i}]_1^m \\ \text{old death pred. proofs} & [\pi_{\mathsf{d},i}]_1^m \\ \text{new birth pred. ver. keys} & [\mathsf{vk}_{\mathsf{b},j}]_1^n \\ \text{new birth pred. proofs} & [\pi_{\mathsf{b},j}]_1^n \\ \text{predicate comm. randomness} & r_\Phi \end{pmatrix}$$

A witness $\mathbb{w}_{\mathsf{CP}}$ is valid for an instance $\mathbb{x}_{\mathsf{CP}}$ if the following conditions hold:

1. For each $i \in \{i, \dots, m\}$:
   - The death predicate hash $h_{\mathsf{d},i}$ is computed correctly: $h_{\mathsf{d},i} = \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{vk}_{\mathsf{d},i})$.
   - The death predicate proof $\pi_{\mathsf{d},i}$ is valid: $\mathsf{NIZK.Verify}(\mathsf{vk}_{\mathsf{d},i}, i \| \mathsf{cm}_{\mathsf{ldata}}, \pi_{\mathsf{d},i})$.
2. For each $j \in \{1, \dots, n\}$:
   - The birth predicate hash $h_{\mathsf{b},j}$ is computed correctly: $h_{\mathsf{b},j} = \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, \mathsf{vk}_{\mathsf{b},j})$.
   - The birth predicate proof $\pi_{\mathsf{b},j}$ is valid: $\mathsf{NIZK.Verify}(\mathsf{vk}_{\mathsf{b},j}, j \| \mathsf{cm}_{\mathsf{ldata}}, \pi_{\mathsf{b},j})$.
3. The predicate commitment cm_Φ is valid: $\mathsf{cm}_\Phi = \mathsf{b2s}([h_{\mathsf{d},i}]_1^m \ \|\ [h_{\mathsf{b},j}]_1^n \ \|\ r_\Phi)$.

Figure 14: Splitting the NP relation $\mathcal{R}_{\mathsf{e}}$ into two NP relations $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$, over $\mathbb{F}_r$ and $\mathbb{F}_p$ respectively.

8 System implementation

We implemented our "plain" DPC scheme (Section 4) and our delegable DPC scheme (Section 5), by following the strategy described in Section 7. The resulting system, named **Zexe** (*Zero knowledge EXEcution*), consists of several Rust libraries: (a) a library for finite field and elliptic curve arithmetic, adapted from [Bow17b]; (b) a library for cryptographic building blocks, including zkSNARKs for constraint systems (using components from [Bow17a]); (c) a library with constraints for many of these building blocks; and (d) a library that realizes our constructions of plain and delegable DPC. Our code base, like our construction, is written in terms of abstract building blocks, which allows to easily switch between different instantiations of the building blocks. In the rest of this section we describe the efficient instantiations used in the experiments reported in Section 9.

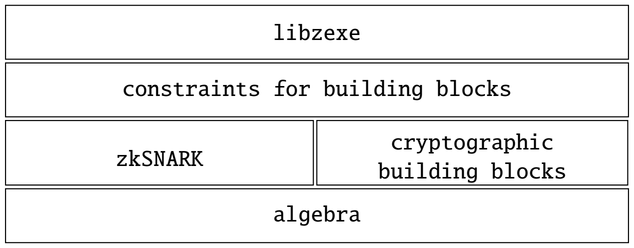
libzexe
constraints for building blocks
zkSNARK
cryptographic building blocks
algebra

Figure 15: Stack of libraries comprising Zexe.

Ledger. The ledger $\mathbf{L}$ in our prototype is simply an ideal ledger, i.e., an append-only log of valid transactions that is stored in memory. Of course, in a real-world deployment, this ideal ledger would be replaced by a distributed protocol that realizes (a suitable approximation of) an ideal ledger. Recall from Section 3.1 that we require the ledger $\mathbf{L}$ to provide a method to efficiently prove and verify membership of a transaction, or one of its subcomponents, in $\mathbf{L}$. For this, we maintain a Merkle tree [Mer87] atop the list of transactions, using the collision-resistant hash function CRH described below. This results in the following algorithms for $\mathbf{L}$.

- $\mathbf{L}.\mathsf{Push}(\mathsf{tx})$: Append tx to the transaction list and update the Merkle tree.
- $\mathbf{L}.\mathsf{Digest} \to \mathsf{st}_{\mathbf{L}}$: Return the root of the Merkle tree.
- $\mathbf{L}.\mathsf{Prove}(\mathsf{tx}) \to \mathbb{w}_{\mathbf{L}}$: Return the authentication path for tx in the Merkle tree.
- $\mathbf{L}.\mathsf{Verify}(\mathsf{st}_{\mathbf{L}}, \mathsf{tx}, \mathbb{w}_{\mathbf{L}}) \to b$: Check that $\mathbb{w}_{\mathbf{L}}$ is a valid authentication path for tx in a tree with root $\mathsf{st}_{\mathbf{L}}$.

Our prototype maintains the Merkle tree in memory, but a real-world deployment would have to maintain it via a distributed protocol. (Such data structures atop distributed ledgers are used in existing systems [ZCa15].)

Pseudorandom function. Fixing key length and input length at 256 bits, we instantiate PRF using the Blake2s hash function [ANWW13]: $\mathsf{PRF}_k(x) := \mathsf{b2s}(k\|x)$ for $k, x \in \{0,1\}^{256}$.

Elliptic curves. Our implementation strategy (see Section 7) involves several elliptic curves: two pairing-friendly curves E_{BLS} and E_{CP}, and two "plain" curves $E_{\mathsf{Ed/BLS}}$ and $E_{\mathsf{Ed/CP}}$ whose base field respectively matches the prime-order subgroup of E_{BLS} and E_{CP}. Details about these curves are in Figure 16; the parameter used to generate the BLS curve E_{BLS} is $x = 3 \cdot 2^{46} \cdot (7 \cdot 13 \cdot 499) + 1$ (see Section 7 for why).

NIZKs. We instantiate the NIZKs used for the NP relation $\mathcal{R}_{\mathsf{e}}$ via zero-knowledge *succinct* non-interactive arguments of knowledge (zk-SNARKs), which makes our DPC schemes succinct (see Remark 4.1). Concretely, we rely on the simulation-extractable zkSNARK of Groth and Maller [GM17], used over the pairing-friendly elliptic curves E_{BLS} (for proving predicates' satisfiability) and E_{CP} (for proving validity of these latter proofs).

DLP-hard group. Several instantiations of cryptographic primitives introduced below rely on the hardness of extracting discrete logarithms in a prime order group. We generate these groups via a group generator $\mathsf{SampleGrp}$, which on input a security parameter λ (represented in unary), outputs a tuple $(\mathbb{G}, q, g)$ that

name	curve type	embedding degree	size of prime-order subgroup	size of base field	size of compressed group elements (rounded to multiples of 8 bytes)	
					$\mathbb{G}_1$	$\mathbb{G}_2$
$E_{\mathsf{Ed/BLS}}$	twisted Edwards	—	s	r	32	—
E_{BLS}	BLS	12	r	p	48	96
$E_{\mathsf{Ed/CP}}$	twisted Edwards	—	t	p	48	—
E_{CP}	short Weierstrass	6	p	q	104	312

prime	value	size in bits	2-adicity
s	`0x4aad957a68b2955982d1347970dec005293a3afc43c8afeb` `95aee9ac33fd9ff`	251	1
r	`0x12ab655e9a2ca55660b44d1e5c37b00159aa76fed0000001` `0a11800000000001`	253	47
t	`0x35c748c2f8a21d58c760b80d94292763445b3e601ea271e1` `d75fe7d6eeb84234066d10f5d893814103486497d95295`	374	2
p	`0x1ae3a4617c510eac63b05c06ca1493b1a22d9f300f5138f1` `ef3622fba094800170b5d44300000008508c00000000001`	377	46
q	`0x3848c4d2263babf8941fe959283d8f526663bc5d176b746a` `f0266a7223ee72023d07830c728d80f9d78bab3596c8617c57` `9252a3fb77c79c13201ad533049cfe6a399c2f764a12c4024b` `ee135c065f4d26b7545d85c16dfd424adace79b57b942ae9`	782	3

Figure 16: The elliptic curves $E_{\mathsf{BLS}}, E_{\mathsf{CP}}, E_{\mathsf{Ed/BLS}}, E_{\mathsf{Ed/CP}}$.

describes a group $\mathbb{G}$ of prime order q generated by g. The discrete-log problem is hard in $\mathbb{G}$. In our prototype we fix $\mathbb{G}$ to be the largest prime-order subgroup of either $E_{\mathsf{Ed/BLS}}$ or $E_{\mathsf{Ed/CP}}$, depending on the context.

Trapdoor commitments. We instantiate trapdoor commitments via Pedersen commitments over $\mathbb{G}$, as defined in Figure 17; note that the setup algorithm takes as additional input the message length n. Pedersen commitments are perfectly hiding, and are computationally binding if the discrete-log problem is hard in $\mathbb{G}$.

Collision-resistant hashing. We instantiate CRH via a Pedersen hash function over $\mathbb{G}$, as specified in Figure 18. Note that the setup algorithm takes as additional input the message length n. Collision resistance follows from hardness of the discrete-logarithm problem [MRK03].

Remark 8.1. Hopwood et al. [HBHW18] note that projecting a twisted Edwards curve point (x, y) to its x-coordinate is injective when the point is in the curve's largest prime-order subgroup. Our implementation uses this fact to reduce the output size of TCM and CRH by projecting their output to its x-coordinate.

```
$\mathsf{TCM.Setup}(1^\lambda, n) \to \mathsf{pp}_{\mathsf{TCM}}$:
1. Sample a group: $(\mathbb{G}, q, g) \leftarrow \mathsf{SampleGrp}(1^\lambda)$.
2. For $i \in \{1, \ldots, n\}$, sample generator $h_i$:
   $r_i \leftarrow \mathbb{Z}_q; h_i := g^{r_i}$.
3. Output $\mathsf{pp}_{\mathsf{TCM}} := (\mathbb{G}, q, g, [h_i]_1^n)$.

$\mathsf{TCM.Commit}(\mathsf{pp}_{\mathsf{TCM}}, m \in \{0,1\}^n; r_{\mathsf{cm}}) \to \mathsf{cm}$:
1. Parse $\mathsf{pp}_{\mathsf{TCM}}$ as $(\mathbb{G}, q, g, [h_i]_1^n)$.
2. Output $\mathsf{cm} := g^{r_{\mathsf{cm}}} \prod_{i=1}^n h_i^{m_i}$.
```

Figure 17: Pedersen commitment scheme.

```
$\mathsf{CRH.Setup}(1^\lambda, n) \to \mathsf{pp}_{\mathsf{CRH}}$:
1. Sample a group: $(\mathbb{G}, q, g_1) \leftarrow \mathsf{SampleGrp}(1^\lambda)$.
2. For $i \in \{2, \ldots, n\}$, sample generator $g_i$:
   $r_i \leftarrow \mathbb{Z}_q; g_i := g^{r_i}$.
3. Output $\mathsf{pp}_{\mathsf{CRH}} := (\mathbb{G}, q, [g_i]_1^n)$.

$\mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, m \in \{0,1\}^n) \to h$:
1. Parse $\mathsf{pp}_{\mathsf{CRH}}$ as $(\mathbb{G}, q, [g_i]_1^n)$.
2. Output $h := \prod_{i=1}^n g_i^{m_i}$.
```

Figure 18: Pedersen collision-resistant hash.

9 System evaluation

In Section 9.1 we evaluate individual cryptographic building blocks. In Section 9.2 we evaluate the cost of NP relations expressed as constraints, as required by the underlying zkSNARK. In Section 9.3 we evaluate the running time of DPC algorithms. In Section 9.4 we evaluate the sizes of DPC data structures. All reported measurements were taken on a machine with an Intel Xeon 6136 CPU at $3.0\,\mathrm{GHz}$ with $252\,\mathrm{GB}$ of RAM.

9.1 Cryptographic building blocks

We are interested in two types of costs associated with a given cryptographic building block: the *native execution cost*, which are the running times of certain algorithms on a CPU; and the *constraint cost*, which are the numbers of constraints required to express certain invariants, to be used by the underlying zkSNARK.

Native execution cost. The zkSNARK dominates native execution cost, and the costs of all other building blocks are negligible in comparison. Therefore we separately report only the running times of the zkSNARK, which in our case is a protocol due to Groth and Maller [GM17], abbreviated as GM17. When instantiated over the elliptic curve E_{BLS}, the GM17 prover takes $25\,\mu\mathrm{s}$ per constraint (with 12 threads), while the GM17 verifier takes $250\,n\,\mu\mathrm{s} + 9.5\,\mathrm{ms}$ on an input with n field elements (with 1 thread). When instantiated over the elliptic curve E_{CP}, the respective prover and verifier costs are $147\,\mu\mathrm{s}$ per constraint and $1.6\,n\,\mathrm{ms}$ + $34\,\mathrm{ms}$.

Constraint cost. There are three building blocks that together account for the majority of the cost of NP statements that we use. These are: (a) the Blake2s PRF, which requires 21792 constraints to map a 64-byte input to a 32-byte output; (b) the Pedersen collision-resistant hash, which requires $5n$ constraints for an input of n bits; and (c) the GM17 verifier, which requires $14n + 52626$ constraints for an n-bit input.

9.2 The execute NP relation

In many zkSNARK constructions, including the one that we use, one must express all the relevant checks in the given NP relation as (rank-1) *quadratic constraints* over a certain large prime field. The goal is to minimize the number of such constraints because the prover's costs grow (quasi)linearly in this number.

In our DPC scheme we use a zkSNARK for the NP relation $\mathcal{R}_{\mathsf{e}}$ in Fig. 9 and, similarly, in our delegable DPC scheme we use it for the NP relation $\mathcal{R}_{\mathsf{e}}^{\mathsf{del}}$ in Fig. 23. More precisely, for efficiency reasons explained in Section 7, we split $\mathcal{R}_{\mathsf{e}}$ into the two NP relations $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$ in Fig. 14, which we prove via zkSNARKs over the pairing-friendly curves E_{BLS} and E_{CP}, respectively. (We also similarly split $\mathcal{R}_{\mathsf{e}}^{\mathsf{del}}$.)

Table 3 reports the number of constraints that that we use to express $\mathcal{R}_{\mathsf{BLS}}$, as a function of the number of input (m) and output (n) records, and additionally reports its primary contributors. Table 4 does the same for $\mathcal{R}_{\mathsf{CP}}$. These tables show that for each input record costs are dominated by verification of a Merkle tree path and the verification of a (death predicate) proof; while for each output record costs are dominated by the verification of a (birth predicate) proof.

9.3 DPC algorithms

In Table 1 we report the running times of algorithms in our plain DPC and delegable DPC implementations for two input and two output records. Note that for $\mathsf{Execute}$ and Verify, we have excluded costs of ledger operations (such as retrieving an authentication path or scanning for duplicate serial numbers) because these depend on how a ledger is realized, which is orthogonal to our work. Also, we assume that $\mathsf{Execute}$ receives as inputs the SNARK proofs checked by the NP relation. Producing each of these proofs requires invoking the

GM17 prover, over the elliptic curve E_{BLS}, for the relevant birth or death predicate; we provide the amortized time per constraint for this in Section 9.1.

Observe that the overhead incurred by delegable DPC over plain DPC is negligible, and that, as expected, Setup and Execute are the most costly algorithms, as they invoke costly zkSNARK setup and proving algorithms. To mitigate these costs, Setup and Execute are executed on 12 threads; everything else is executed with 1 thread. Overall, we learn that Execute takes less than 2 minutes, and Verify takes tens of milliseconds.

9.4 DPC data structures

Addresses. An address public key in a DPC scheme is a point on the elliptic curve $E_{\mathsf{Ed/BLS}}$, which is 32 bytes when compressed (see Fig. 16); the corresponding secret key is 96 bytes and consists of a PRF seed (32 bytes), address metadata (32 bytes), and commitment randomness (32 bytes). In a delegable DPC scheme, address public keys do not change, but address secret keys are 128 bytes, because they additionally contain the 32-byte secret key of a randomizable signature scheme over the elliptic curve $E_{\mathsf{Ed/BLS}}$ (see Fig. 10).

Transactions. A transaction in a DPC scheme, with two input and two output records, is 968 bytes. It contains two zkSNARK proofs: π_{BLS}, over the elliptic curve E_{BLS}, and π_{CP}, over the curve E_{CP}. Each proof consists of two $\mathbb{G}_1$ and one $\mathbb{G}_2$ elements from its respective curve, amounting to 192 bytes for π_{BLS} and 520 for π_{CP} (both in compressed form). In general, for m input records and n output records, transactions are $32m + 32n + 840$ bytes. In a delegable DPC scheme, a transaction additionally contains a 64-byte signature for each input record. See Table 2 for a detailed break down of all of these costs.

Record contents. We set a record's payload to be 32 bytes long; if a predicate needs longer data then it can set the payload to be the hash of this data, and use non-determinism to access the data. The foregoing choice means that all contents of a record add up to 224 bytes, since a record consists of an address public key (32 bytes), the 32-byte payload, hashes of birth and death predicates (48 bytes each), a serial number nonce (32 bytes), and commitment randomness (32 bytes).

	Plain DPC	Delegable DPC
Setup	188.63 s	187.76 s
GenAddress	14 ms	24 ms
Execute	86.9 s	87.02 s
Verify	46 ms	69 ms

Table 1: Cost of DPC algorithms for 2 inputs and 2 outputs.

	Plain DPC	Delegable DPC
2 inputs and 2 outputs	968	1096
m inputs and n outputs	$32m + 32n + 840$	$96m + 32n + 840$
Per input record:		
Serial number	32	32
Signature	—	64
Per output record:		
Commitment	32	32
Memorandum	32	32
zkSNARK proof over E_{CP}	520	520
zkSNARK proof over E_{BLS}	192	192
Predicate commitment	32	32
Local data commitment	32	32
Ledger digest	32	32

Table 2: Size of a DPC transaction (in bytes)

		Plain DPC	Delegable DPC
Total with 2 inputs and 2 outputs		$\mathbf{454122}$	$\mathbf{467737}$
Below we provide a breakdown of the number of constraints with m input and n output records.			
Per input record	Total	141045	147211
	Enforce validity of:		
	Merkle tree path	98208	98208
	Address key pair	6663	9740
	Serial number computation	22301	25390
	Record commitment	13873	13873
Per output record	Total	20828	20828
	Enforce validity of:		
	Serial number nonce	6697	6697
	Record commitment	14131	14131
Other:	Enforce validity of:		
	Predicate commitment	$21792 \cdot \lceil \frac{3}{4}(m+n) + \frac{1}{2} \rceil$	$21792 \cdot \lceil \frac{3}{4}(m+n) + \frac{1}{2} \rceil$
	Local data commitment	$10240 \cdot m + 7680 \cdot n$	$10240 \cdot m + 7680 \cdot n$
	Miscellaneous	7368	8651

Table 3: Number of constraints for $\mathcal{R}_{\mathsf{BLS}}$.

		Plain DPC	Delegable DPC
Total with 2 inputs and 2 outputs		$\mathbf{556678}$	$\mathbf{556930}$
Below we provide a breakdown of the number of constraints with m input and n output records.			
Per input record	Total	116902	116902
	Enforce validity of:		
	Death predicate ver. key	56797	56797
	Death predicate proof	60105	60105
Per output record	Total	116902	116902
	Enforce validity of:		
	Birth predicate ver. key	56797	56797
	Birth predicate proof	60105	60105
Other	Enforce validity of:		
	Predicate commitment	$21792 \cdot \lceil \frac{3}{4}(m+n) + \frac{1}{2} \rceil$	$21792 \cdot \lceil \frac{3}{4}(m+n) + \frac{1}{2} \rceil$
	Miscellaneous	1902	2154

Table 4: Number of constraints for $\mathcal{R}_{\mathsf{CP}}$.

A Proof of security for our DPC scheme

We prove that our DPC construction (see Section 4) satisfies the security definition in Section 3.3. To do this, for every real-world (efficient) adversary $\mathcal{A}$, we construct an ideal-world (efficient) simulator $\mathcal{S}$ such that the ideal-world and real-world executions are computationally indistinguishable with respect to any (efficient) environment $\mathcal{E}$. We proceed in three parts: in Appendix A.1 we describe building blocks used to construct the simulator $\mathcal{S}$; in Appendix A.2 we describe the simulator $\mathcal{S}$; in Appendix A.3 we argue that the ideal-world and the real-world executions are computationally indistinguishable.

A.1 Building blocks for the simulator

We describe various algorithms that are used as sub-routines in the simulator $\mathcal{S}$.

Trapdoor commitments. Recall from Section 4.1 that a *trapdoor* commitment scheme is a commitment scheme with auxiliary algorithms $(\mathsf{SimSetup}, \mathsf{Equivocate})$ that enable one to open a commitment cm to any chosen message. Below we restrict cm to be a commitment to the empty string ε because this is sufficient for the proof of security of our DPC scheme.

- *Trapdoor setup:* on input a security parameter, $\mathsf{TCM.SimSetup}$ samples public parameters $\mathsf{pp}_{\mathsf{TCM}}$ and a trapdoor $\mathsf{td}_{\mathsf{TCM}}$ such that $\mathsf{pp}_{\mathsf{TCM}}$ is indistinguishable from public parameters sampled by $\mathsf{TCM.Setup}$.
- *Equivocation:* on input public parameters $\mathsf{pp}_{\mathsf{TCM}}$, trapdoor $\mathsf{td}_{\mathsf{TCM}}$, commitment cm to ε, corresponding commitment randomness r_{cm} (so that $\mathsf{TCM.Commit}(\mathsf{pp}_{\mathsf{TCM}}, \varepsilon; r_{\mathsf{cm}}) = \mathsf{cm}$), and target message m', $\mathsf{TCM.Equivocate}$ outputs commitment randomness r'_{cm} such that $\mathsf{TCM.Commit}(\mathsf{pp}_{\mathsf{TCM}}, m'; r'_{\mathsf{cm}}) = \mathsf{cm}$. Moreover, if r_{cm} is uniformly random then r'_{cm} is statistically close to uniformly random.

In Figure 19 we instantiate these algorithms for the Pedersen commitment scheme. Note that the real and simulated public parameters are identical; moreover, the trapdoor randomness r'_{cm} is the real randomness r_{cm} shifted by uniformly random field elements, and is hence statistically close to r_{cm}.

```
$\mathsf{TCM.SimSetup}(1^\lambda, n) \to (\mathsf{pp}_{\mathsf{TCM}}, \mathsf{td}_{\mathsf{TCM}})$
1. Sample a group: $(\mathbb{G}, q, g) \leftarrow \mathsf{SampleGrp}(1^\lambda)$.
2. For $i \in \{1, \ldots, n\}$:
     sample $r_i$ uniformly from $\mathbb{Z}_q$, and set $h_i := g^{r_i}$.
3. Output $(\mathsf{pp}_{\mathsf{TCM}} := (\mathbb{G}, q, g, [h_i]_1^n), \mathsf{td}_{\mathsf{TCM}} := [r_i]_1^n)$.
```

```
$\mathsf{TCM.Equivocate}(\mathsf{pp}_{\mathsf{TCM}}, \mathsf{td}_{\mathsf{TCM}}, \mathsf{cm}, r_{\mathsf{cm}}, m' \in \{0,1\}^n) \to r'_{\mathsf{cm}}$
1. Parse $\mathsf{pp}_{\mathsf{TCM}}$ as $(\mathbb{G}, q, g, [h_i]_1^n)$.
2. Parse $\mathsf{td}_{\mathsf{TCM}}$ as $[r_i]_1^n$.
3. Output $r'_{\mathsf{cm}} := r_{\mathsf{cm}} - \sum_{i=1}^n r_i m'_i \bmod q$.
```

Figure 19: Simulated setup and equivocation algorithms for the Pedersen commitment scheme.

NIZKs. The scheme $\mathsf{NIZK} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ is a *simulation-extractable* non-interactive zero knowledge argument. Formally stating the properties of this scheme involves several auxiliary algorithms.
– *Trapdoor setup:* on input a security parameter and a description of an NP relation $\mathcal{R}$, $\mathsf{NIZK.SimSetup}$ outputs a set of public parameters $\mathsf{pp}_{\mathsf{NIZK}}$ and a trapdoor $\mathsf{td}_{\mathsf{NIZK}}$.
– *Simulation:* on input public parameters $\mathsf{pp}_{\mathsf{NIZK}}$, trapdoor $\mathsf{td}_{\mathsf{NIZK}}$, NP instance $\mathbb{x}$, and (optionally) auxiliary information aux, $\mathsf{NIZK.Simulate}$ outputs a simulated proof π.
– *Extraction:* on input public parameters $\mathsf{pp}_{\mathsf{NIZK}}$, trapdoor $\mathsf{td}_{\mathsf{NIZK}}$, NP instance $\mathbb{x}$, and proof π, $\mathsf{NIZK.Extract}$ outputs a witness $\mathbb{w}$ such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ (allegedly).
We can now state the properties satisfied by NIZK.

- *Completeness:* for every NP relation $\mathcal{R}$ and instance-witness pair $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$,

$$\Pr\left[\mathsf{NIZK.Verify}(\mathsf{pp}_{\mathsf{NIZK}}, \mathbb{x}, \pi) = 1 \;\middle|\; \begin{array}{c} \mathsf{pp}_{\mathsf{NIZK}} \leftarrow \mathsf{NIZK.Setup}(1^\lambda, \mathcal{R}) \\ (\mathbb{x}, \pi) \leftarrow \mathsf{NIZK.Prove}(\mathsf{pp}_{\mathsf{NIZK}}, \mathbb{x}, \mathbb{w}) \end{array} \right] = 1 \ .$$

- *Perfect zero knowledge:* for every relation $\mathcal{R}$ and efficient adversary $\mathcal{A}$,

$$\Pr\left[\begin{array}{l}\mathsf{pp_{NIZK}} \leftarrow \mathsf{NIZK.Setup}(1^\lambda, \mathcal{R}) \\ \mathcal{A}^{S_1(\cdot,\cdot)}(\mathsf{pp_{NIZK}}, \mathsf{aux}) = 1\end{array}\right] = \Pr\left[\begin{array}{l}(\mathsf{pp_{NIZK}}, \mathsf{td_{NIZK}}) \leftarrow \mathsf{NIZK.SimSetup}(1^\lambda, \mathcal{R}) \\ \mathcal{A}^{S_2(\cdot,\cdot)}(\mathsf{pp_{NIZK}}, \mathsf{aux}) = 1\end{array}\right]$$

  where the two oracles are defined as follows
  - $S_1(\mathbb{x}, \mathbb{w})$:= "if $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ then $\mathsf{NIZK.Prove}(\mathsf{pp_{NIZK}}, \mathbb{x}, \mathbb{w})$, else abort";
  - $S_2(\mathbb{x}, \mathbb{w})$:= "if $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ then $\mathsf{NIZK.Simulate}(\mathsf{pp_{NIZK}}, \mathsf{td_{NIZK}}, \mathbb{x})$, else abort".
- *Simulation extractability:* for every relation $\mathcal{R}$ and efficient adversary $\mathcal{A}$,

$$\Pr\left[\begin{array}{l|r}(\mathbb{x}, \pi) \notin Q & (\mathsf{pp_{NIZK}}, \mathsf{td_{NIZK}}) \leftarrow \mathsf{NIZK.SimSetup}(1^\lambda, \mathcal{R}) \\ (\mathbb{x}, \mathbb{w}) \notin \mathcal{R} & (\mathbb{x}, \pi) \leftarrow \mathcal{A}^{S(\cdot)}(\mathsf{pp_{NIZK}}) \\ \mathsf{NIZK.Verify}(\mathsf{pp_{NIZK}}, \mathbb{x}, \pi) = 1 & \mathbb{w} \leftarrow \mathsf{NIZK.Extract}(\mathsf{pp_{NIZK}}, \mathsf{td_{NIZK}}, \mathbb{x}, \pi)\end{array}\right] = \mathrm{negl}(\lambda) \quad ,$$

  where $S(\mathbb{x}) := \mathsf{NIZK.Simulate}(\mathsf{pp_{NIZK}}, \mathsf{td_{NIZK}}, \mathbb{x})$ and Q is the set of query-answer pairs between the adversary $\mathcal{A}$ and the simulated-proof oracle S.

A.2 The ideal-world simulator

The ideal-word simulator $\mathcal{S}$ will interact with the ideal functionality $\mathcal{F}_{\mathrm{DPC}}$ and with the environment $\mathcal{E}$. Note that for UC security it suffices to show security against a dummy real-world adversary $\mathcal{A}$ that simply forwards all instructions from the environment $\mathcal{E}$ [Can01]. Since our security definition is a special case of UC security, we inherit this simplification, and thus only consider such an adversary $\mathcal{A}$. The pseudocode for $\mathcal{S}$ is provided below; auxiliary subroutines are provided in Figure 20.

Setup.

1. Initialize an empty table $\mathcal{S}.\mathsf{Records}$ that maps record commitments to their contents.
2. Initialize an empty table $\mathcal{S}.\mathsf{AddrPk}$ that maps address public keys to their secret keys.
3. Initialize an empty transaction ledger $\mathbf{L}$.
4. Sample simulated public parameters and trapdoor: $(\mathsf{pp}, \mathsf{td}) \leftarrow \mathsf{DPC.SimSetup}(1^\lambda)$. (See Fig. 20.)
5. Define

$$\begin{aligned}\mathsf{SampleAddrPk}(\cdot) &:= \mathsf{TCM.Commit}(\mathsf{pp_{TCM}}, \varepsilon; \cdot) \quad , \\ \mathsf{SampleCm}(\cdot) &:= \mathsf{TCM.Commit}(\mathsf{pp_{TCM}}, \varepsilon; \cdot) \quad , \\ \mathsf{SampleSn}(\cdot) &:= \text{“sample uniformly random string of correct length”} \quad .\end{aligned}$$

6. Start ideal-world execution with the above $(\mathsf{SampleAddrPk}, \mathsf{SampleCm}, \mathsf{SampleSn})$.

At this point, the simulator will receive messages notifying it of transactions and of messages sharing contents of newly-created records. The simulator handles each case separately.

Transaction notifications.

- **From environment.** When $\mathcal{E}$ instructs a corrupted party to invoke $\mathbf{L}.\mathsf{Push}(\mathsf{tx})$:
  1. If $\mathsf{DPC.Verify}^{\mathbf{L}}(\mathsf{pp}, \mathsf{tx}) \neq 1$, abort.
  2. Parse the real-world transaction tx as $([\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo}, \star)$.
  3. Compute $([\mathbf{r}_i]_1^m, [\mathsf{ask}_i]_1^m, [\mathbf{r}_j]_1^n, \mathsf{aux}) \leftarrow \mathsf{DPC.ExtractExecute}(\mathsf{pp}, \mathsf{td}, \mathsf{tx})$. [See Figure 20.]
  4. For every $i \in \{1, \dots, m\}$:

   (a) Parse the real-world record $\mathbf{r}_i$ as $(\mathsf{apk}_i, \mathsf{payload}_i, \Phi_{\mathsf{b},i}, \Phi_{\mathsf{d},i}, \rho_i, r_i, \mathsf{cm}_i)$.
   (b) Parse the address secret key ask_i as $(\mathsf{sk}_{\mathsf{PRF},i}, \mathsf{meta}_i, r_{\mathsf{pk},i})$.
   (c) If $\mathcal{S}.\mathsf{Records}[\mathsf{cm}_i] \neq \mathbf{r}_i$, abort. (***Note:*** *Captures binding property of the commitment.*)
   (d) If $\mathbf{L}.\mathsf{Contains}(\mathsf{cm}_i) = 0$, abort. (***Note:*** *Captures existence of record.*)
   (e) Create the ideal-world record $\mathbb{r}_i := (\mathsf{cm}_i, \mathsf{apk}_i, \mathsf{payload}_i, \Phi_{\mathsf{b},i}, \Phi_{\mathsf{d},i})$.
   (f) If $\mathcal{S}.\mathsf{AddrPk}[\mathsf{apk}_i] = \perp$:
      i. Invoke $\mathcal{F}_{\text{DPC}}.\mathsf{GenAddress}(\mathsf{meta}_i, \mathsf{apk}_i)$.
      ii. Insert apk_i into $\mathcal{S}.\mathsf{AddrPk}$: $\mathcal{S}.\mathsf{AddrPk}[\mathsf{apk}_i] := \mathsf{ask}_i$.
   (g) Else, if $\mathcal{S}.\mathsf{AddrPk}[\mathsf{apk}_i] \neq \mathsf{ask}_i$, abort. (***Note:*** *Captures uniqueness of secret key.*)
5. For every $j \in \{1, \ldots, n\}$:
   (a) Parse the real-world record $\mathbf{r}_j$ as $(\mathsf{apk}_j, \mathsf{payload}_j, \Phi_{\mathsf{b},j}, \Phi_{\mathsf{d},j}, \rho_j, r_j, \mathsf{cm}_j)$.
   (b) If the serial number nonce ρ_j was seen in a prior extracted transaction, or if $\rho_j = \rho_k$ for $k \neq j$, abort. (***Note:*** *Captures uniqueness of nonce.*)
   (c) Set $\mathcal{S}.\mathsf{Records}[\mathsf{cm}_j] := \mathbf{r}_j$.
6. Construct instance for $\mathcal{R}_\mathsf{e}$: $\mathbb{x}_\mathsf{e} := (\mathsf{st}_\mathbf{L}, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo})$.
7. Construct witness for $\mathcal{R}_\mathsf{e}$: $\mathbb{w}_\mathsf{e} := ([\mathbf{r}_i]_1^m, [\mathbb{w}_{\mathbf{L},i}]_1^m, [\mathsf{ask}_i]_1^m, [\mathbf{r}_j]_1^n, \mathsf{aux})$.
8. If $(\mathbb{x}_\mathsf{e}, \mathbb{w}_\mathsf{e}) \notin \mathcal{R}_\mathsf{e}$, abort.
9. Invoke $\mathcal{F}_{\text{DPC}}.\mathsf{Execute}([\mathbb{r}_i]_1^m, [\mathsf{meta}_i]_1^m, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, [\mathsf{apk}_j]_1^n, [\mathsf{payload}_j]_1^n, [\Phi_{\mathsf{b},j}]_1^n, [\Phi_{\mathsf{d},j}]_1^n, \mathsf{aux}, \mathsf{memo})$.
10. **Receive from $\mathcal{F}_{\text{DPC}}$:** $[\mathbb{r}_j]_1^n$.
11. **Receive from $\mathcal{F}_{\text{DPC}}$:** $(\mathtt{Execute}, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo})$.
12. Append the real-world transaction tx to the ledger $\mathbf{L}$.

- **From ideal functionality.** When $\mathcal{F}_{\text{DPC}}$ broadcasts $(\mathtt{Execute}, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo})$:
  1. Compute $([\mathbf{r}_j]_1^n, \mathsf{tx}) \leftarrow \mathsf{DPC.SimExecute}^\mathbf{L}(\mathsf{pp}, \mathsf{td}, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo})$. (See Fig. 20.)
  2. For each $j \in \{1, \ldots, n\}$, set $\mathcal{S}.\mathsf{Records}[\mathsf{cm}_j] := \mathbf{r}_j$.
  3. Append the real-world transaction tx to the ledger $\mathbf{L}$.

Record authorization notification.

- **From environment.** When $\mathcal{E}$ instructs a corrupted party to send $(\mathtt{RecordAuth}, \mathbf{r}, \mathcal{P})$ to $\mathcal{P}$:
  1. Parse the real-world record $\mathbf{r}$ as $(\mathsf{apk}, \mathsf{payload}, \Phi_\mathsf{b}, \Phi_\mathsf{d}, \rho, r, \mathsf{cm})$.
  2. Invoke $\mathcal{F}_{\text{DPC}}.\mathsf{ShareRecord}(\mathbb{r}, \mathcal{P})$ with $\mathbb{r} := (\mathsf{cm}, \mathsf{apk}, \mathsf{payload}, \Phi_\mathsf{b}, \Phi_\mathsf{d})$.
- **From ideal functionality.** When $\mathcal{F}_{\text{DPC}}$ sends $(\mathtt{RecordAuth}, \mathbb{r}, r)$:
  1. Parse the ideal record $\mathbb{r}$ as $(\mathsf{cm}, \mathsf{apk}, \mathsf{payload}, \Phi_\mathsf{b}, \Phi_\mathsf{d})$.
  2. Retrieve the real-world record $\mathbf{r} = \mathcal{S}.\mathsf{Records}[\mathsf{cm}]$, and set the serial number nonce $\rho := \mathbf{r}.\rho$.
  3. Define new record commitment message $m := (\mathsf{apk} \| \mathsf{payload} \| \Phi_\mathsf{b} \| \Phi_\mathsf{d} \| \rho)$.
  4. Compute new commitment randomness $r' \leftarrow \mathsf{TCM.Equivocate}(\mathsf{pp}_\mathsf{TCM}, \mathsf{td}_\mathsf{TCM}, \mathsf{cm}, r, m)$.
  5. Construct the new real-world record $\mathbf{r}' := (\mathsf{apk}, \mathsf{payload}, \Phi_\mathsf{b}, \Phi_\mathsf{d}, \rho, r', \mathsf{cm})$.
  6. Set $\mathcal{S}.\mathsf{Records}[\mathsf{cm}] := \mathbf{r}'$.
  7. **Send to $\mathcal{A}$:** $(\mathtt{RecordAuth}, \mathbf{r}')$.

```
DPC.SimSetup
Input: security parameter $1^\lambda$
Output: simulated public parameters pp and trapdoor td

1. Sample simulated parameters for trapdoor commitment: $(\mathsf{pp}_{\mathsf{TCM}}, \mathsf{td}_{\mathsf{TCM}}) \leftarrow \mathsf{TCM.SimSetup}(1^\lambda)$.
2. Sample parameters for CRH: $\mathsf{pp}_{\mathsf{CRH}} \leftarrow \mathsf{CRH.Setup}(1^\lambda)$.
3. Sample simulated parameters for NIZK for $\mathcal{R}_\mathsf{e}$: $(\mathsf{pp}_\mathsf{e}, \mathsf{td}_\mathsf{e}) \leftarrow \mathsf{NIZK.SimSetup}(1^\lambda, \mathcal{R}_\mathsf{e})$.
4. Set $\mathsf{pp} := (\mathsf{pp}_{\mathsf{TCM}}, \mathsf{pp}_{\mathsf{CRH}}, \mathsf{pp}_\mathsf{e})$.
5. Set $\mathsf{td} := (\mathsf{td}_{\mathsf{TCM}}, \mathsf{td}_\mathsf{e})$.
6. Output $(\mathsf{pp}, \mathsf{td})$.
```

```
DPC.SimExecute^L
Input:
  • public parameters pp and trapdoor td
  • old serial numbers $[\mathsf{sn}_i]_1^m$
  • new record commitments $[\mathsf{cm}_j]_1^n$
  • transaction memorandum memo
Output: new records $[\mathbf{r}_j]_1^n$ and transaction tx

1. For $j \in \{1, \ldots, n\}$:
    (a) Set new serial number nonce $\rho_j := \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, j \| \mathsf{sn}_1 \| \ldots \| \mathsf{sn}_m)$.
    (b) Set address public key, payload, predicates, and commitment randomness to be the empty string:
        $\mathsf{apk}_j, \mathsf{payload}_j, \Phi_{\mathsf{b},j}, \Phi_{\mathsf{d},j}, r_j := \varepsilon$.
    (c) Construct dummy record: $\mathbf{r}_j := (\mathsf{apk}_j, \mathsf{payload}_j, \Phi_{\mathsf{b},j}, \Phi_{\mathsf{d},j}, \rho_j, r_j, \mathsf{cm}_j)$.
2. Retrieve current ledger digest: $\mathsf{st}_\mathbf{L} \leftarrow \mathbf{L}.\mathsf{Digest}$.
3. Construct instance for relation $\mathcal{R}_\mathsf{e}$: $\mathbb{x}_\mathsf{e} := (\mathsf{st}_\mathbf{L}, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo})$.
4. Generate simulated proof for $\mathcal{R}_\mathsf{e}$: $\pi_\mathsf{e} \leftarrow \mathsf{NIZK.Simulate}(\mathsf{pp}_\mathsf{e}, \mathsf{td}_\mathsf{e}, \mathbb{x}_\mathsf{e})$.
5. Construct transaction: $\mathsf{tx} := ([\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo}, \star)$, where $\star := (\mathsf{st}_\mathbf{L}, \pi_\mathsf{e})$.
6. Output $([\mathbf{r}_j]_1^n, \mathsf{tx})$.
```

```
DPC.ExtractExecute
Input:
  • public parameters pp and trapdoor td
  • transaction tx
Output:
  • old { records $[\mathbf{r}_i]_1^m$
        { address secret keys $[\mathsf{ask}_i]_1^m$
  • new records $[\mathbf{r}_j]_1^n$
  • auxiliary predicate input aux

1. Parse tx as $([\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo}, \star)$ and $\star$ as $(\mathsf{st}_\mathbf{L}, \pi_\mathsf{e})$.
2. Construct instance for relation $\mathcal{R}_\mathsf{e}$: $\mathbb{x}_\mathsf{e} := (\mathsf{st}_\mathbf{L}, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo})$.
3. Obtain witness: $\mathbb{w}_\mathsf{e} \leftarrow \mathsf{NIZK.Extract}(\mathsf{pp}_\mathsf{e}, \mathsf{td}_\mathsf{e}, \mathbb{x}_\mathsf{e}, \pi_\mathsf{e})$.
4. Parse the witness $\mathbb{w}_\mathsf{e}$ as $([\mathbf{r}_i]_1^m, [\mathbb{w}_{\mathbf{L},i}]_1^m, [\mathsf{ask}_i]_1^m, [\mathbf{r}_j]_1^n, \mathsf{aux})$.
5. Output $([\mathbf{r}_i]_1^m, [\mathsf{ask}_i]_1^m, [\mathbf{r}_j]_1^n, \mathsf{aux})$.
```

Figure 20: Several subroutines used by the ideal-world simulator $\mathcal{S}$.

A.3 Proof of security by hybrid argument

We use a sequence of hybrids, each identified by a game $\mathcal{G}_i$, to prove that the outputs of the environment $\mathcal{E}$ when interacting with the real-world (dummy) adversary $\mathcal{A}$ and the ideal-world simulator $\mathcal{S}$ are computationally indistinguishable. We denote by $\mathsf{Output}_i(\mathcal{E})$ the output of $\mathcal{E}$ in game $\mathcal{G}_i$, and by $\mathcal{G}_0$ the real-world execution.

- $\mathcal{G}_1$ (sample parameters):
  This game is the real-world execution modified as follows.
  - $\mathcal{E}$ interacts with $\mathcal{S}$ instead of $\mathcal{A}$.
  - $\mathcal{S}$ uses $\mathsf{DPC.Setup}$ to generate public parameters pp, and gives these to $\mathcal{E}$.
  - $\mathcal{S}$ maintains the ledger $\mathbf{L}$ for $\mathcal{E}$ (it appends to $\mathbf{L}$ any pushed transaction passing the checks in $\mathsf{DPC.Verify}$).
  - $\mathcal{S}$ forwards messages from $\mathcal{E}$ to $\mathbf{L}$ and other parties.
  - $\mathcal{S}$ forwards messages from other honest parties to $\mathcal{E}$.

  $\mathsf{Output}_1(\mathcal{E})$ is perfectly indistinguishable from $\mathsf{Output}_0(\mathcal{E})$ since $\mathcal{S}$ samples the public parameters honestly, maintains the ledger identically to the ideal ledger, and otherwise behaves like the dummy adversary.

- $\mathcal{G}_2$ (simulate setup):
  $\mathcal{S}$ invokes $\mathsf{DPC.SimSetup}$ instead of $\mathsf{DPC.Setup}$. $\mathsf{Output}_2(\mathcal{E})$ is perfectly indistinguishable from $\mathsf{Output}_1(\mathcal{E})$ since NIZK is perfect zero knowledge.

- $\mathcal{G}_3$ (simulate proofs):
  In all honest party transactions, $\mathcal{S}$ replaces NIZK proofs with simulated proofs produced via $\mathsf{NIZK.Simulate}$. $\mathsf{Output}_3(\mathcal{E})$ is perfectly indistinguishable from $\mathsf{Output}_2(\mathcal{E})$ since NIZK is perfect zero knowledge.

- $\mathcal{G}_4$ (simulate serial numbers):
  In all honest party transactions, $\mathcal{S}$ replaces all serial numbers with uniformly random elements sampled from PRF's codomain. Since PRF is a pseudorandom function, and $\mathcal{E}$ does not know the secret key used to compute it, $\mathsf{Output}_4(\mathcal{E})$ is computationally indistinguishable from $\mathsf{Output}_3(\mathcal{E})$.

- $\mathcal{G}_5$ (simulate commitments and equivocate commitment openings):
  In all honest party transactions, $\mathcal{S}$ replaces record commitments with commitments to the empty string ε. In all messages from honest parties to corrupted parties containing record contents, $\mathcal{S}$ replaces the actual commitment randomness with randomness produced by $\mathsf{TCM.Equivocate}$. $\mathsf{Output}_5(\mathcal{E})$ is perfectly indistinguishable from $\mathsf{Output}_4(\mathcal{E})$ since TCM is perfectly hiding and equivocation produces commitment randomness that is statistically close to uniform.

- $\mathcal{G}_6$ (handle adversarial transactions):
  For every corrupted party transaction, $\mathcal{S}$ extracts an NP instance $\mathbb{x}_\mathsf{e}$ and witness $\mathbb{w}_\mathsf{e}$ for $\mathcal{R}_\mathsf{e}$ from the included proof and then proceeds as follows.
  - If $(\mathbb{x}_\mathsf{e}, \mathbb{w}_\mathsf{e}) \notin \mathcal{R}$, $\mathcal{S}$ aborts. If NIZK is simulation-extractable, this occurs with negligible probability.
  - For all $i \in \{1, \ldots, m\}$, if the contents of any $\mathbf{r}_i$ are different from those seen in any `RecordAuth` from an honest party or in the output of a previously extracted transaction, $\mathcal{S}$ aborts. If TCM is a binding commitment scheme, then this occurs with negligible probability.
  - For all $i \in \{1, \ldots, m\}$, if the extracted secret key ask_i for apk_i differs from the secret key extracted for apk_i in a prior transaction, $\mathcal{S}$ aborts. If TCM is a binding commitment scheme, then this occurs with negligible probability.

– For all $j \in \{1, \dots, n\}$, if the serial number nonce ρ_j matches one extracted in a prior transaction, $\mathcal{S}$ aborts. If CRH is a collision-resistant hash, then this occurs with negligible probability because the serial number nonce is the output of CRH evaluated (in part) over the serial numbers of the input records. If this input is distinct across two different invocations of CRH, then collision resistance guarantees that a nonce collision happens with negligible probability. Now for the transaction to be valid, it must contain serial numbers not seen before on the ledger. Therefore, the inputs to CRH are never repeated.

$\mathsf{Output}_6(\mathcal{E})$ is therefore computationally indistinguishable from $\mathsf{Output}_5(\mathcal{E})$.

The final game is distributed identically to the operation of $\mathcal{S}$ from the point of view of $\mathcal{E}$. We have thus shown that $\mathcal{E}$'s advantage in distinguishing the interaction with $\mathcal{S}$ from the interaction with $\mathcal{A}$ is negligible.

B Construction of a delegable DPC scheme

We provide more details on the delegable DPC scheme discussed in Section 5. First we give details on randomizable signatures (Appendix B.1), and then give pseudocode for the DPC construction (Appendix B.2).

B.1 Definition and construction of a randomizable signature scheme

A *randomizable* signature scheme is a tuple of algorithms $\mathsf{SIG} = (\mathsf{Setup}, \mathsf{Keygen}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{RandPk}, \mathsf{RandSig})$ that enables a party to sign messages, while also allowing randomization of public keys and signatures to prevent linking across multiple signatures.

We have already described the syntax of the scheme's algorithms, and summarized its security properties, in Section 5.2. Now we discuss in more detail the security properties, and the construction used in our code.

Security properties. The signature scheme SIG satisfies the following security properties.

- *Existential unforgeability under randomization (EUR).* For every efficient adversary $\mathcal{A}$, the following probability is negligible:

$$\Pr\left[\begin{array}{c} \left(m^* \notin Q \text{ and } \mathsf{SIG.Verify}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}}, m^*, \sigma^*)\right) \\ \text{or} \\ \left(m^* \notin Q \text{ and } \mathsf{SIG.Verify}(\mathsf{pp_{SIG}}, \hat{\mathsf{pk}}_{\mathsf{SIG}}, m^*, \sigma^*)\right) \end{array} \middle| \begin{array}{r} \mathsf{pp_{SIG}} \leftarrow \mathsf{SIG.Setup}(1^\lambda) \\ (\mathsf{pk_{SIG}}, \mathsf{sk_{SIG}}) \leftarrow \mathsf{SIG.Keygen}(\mathsf{pp_{SIG}}) \\ (m^*, \sigma^*, r^*_{\mathsf{SIG}}) \leftarrow \mathcal{A}^{S(\cdot)}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}}) \\ \hat{\mathsf{pk}}_{\mathsf{SIG}} \leftarrow \mathsf{SIG.RandPk}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}}, r^*_{\mathsf{SIG}}) \end{array}\right]$$

  where $S(m) := \mathsf{SIG.Sign}(\mathsf{pp_{SIG}}, \mathsf{sk_{SIG}}, m)$ and Q are the queries made by $\mathcal{A}$ to the signing oracle S.

- *Unlinkability.* Every efficient adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ has at most negligible advantage in guessing the bit b in the $\mathsf{IND\text{-}RSIG}$ game below.

```
$\mathsf{IND\text{-}RSIG}^{\mathsf{SIG}}_{\mathcal{A}}(1^\lambda)$:
1. Generate public parameters: $\mathsf{pp_{SIG}} \leftarrow \mathsf{SIG.Setup}(1^\lambda)$.
2. Generate key pair: $(\mathsf{pk_{SIG}}, \mathsf{sk_{SIG}}) \leftarrow \mathsf{SIG.Keygen}(\mathsf{pp_{SIG}})$.
3. Obtain message from adversary: $m \leftarrow \mathcal{A}_1^{\mathsf{SIG.Sign}(\mathsf{pp_{SIG}}, \mathsf{sk_{SIG}}, \cdot)}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}})$.
4. Sample a bit $b$ uniformly at random.
5. If $b = 0$:
   (a) Sample new key pair: $(\mathsf{pk}'_{\mathsf{SIG}}, \mathsf{sk}'_{\mathsf{SIG}}) \leftarrow \mathsf{SIG.Keygen}(\mathsf{pp_{SIG}})$.
   (b) Sign message: $\sigma \leftarrow \mathsf{SIG.Sign}(\mathsf{pp_{SIG}}, \mathsf{sk}'_{\mathsf{SIG}}, m)$.
   (c) Set $c := (\mathsf{pk}'_{\mathsf{SIG}}, \sigma)$.
6. If $b = 1$:
   (a) Sign message: $\sigma \leftarrow \mathsf{SIG.Sign}(\mathsf{pp_{SIG}}, \mathsf{sk_{SIG}}, m)$.
   (b) Sample randomness $r_{\mathsf{SIG}}$.
   (c) Randomize public key: $\hat{\mathsf{pk}}_{\mathsf{SIG}} \leftarrow \mathsf{SIG.RandPk}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}}, r_{\mathsf{SIG}})$.
   (d) Randomize signature: $\hat{\sigma} \leftarrow \mathsf{SIG.RandSig}(\mathsf{pp_{SIG}}, \sigma, r_{\mathsf{SIG}})$.
   (e) Set $c := (\hat{\mathsf{pk}}_{\mathsf{SIG}}, \hat{\sigma})$.
7. Output $\mathcal{A}_2^{\mathsf{SIG.Sign}(\mathsf{pp_{SIG}}, \mathsf{sk_{SIG}}, \cdot)}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}}, c)$.
```

- *Injective randomization.* For every efficient adversary $\mathcal{A}$, the following probability is negligible:

$$\Pr\left[\begin{array}{c} r_1 \neq r_2 \\ \mathsf{SIG.RandPk}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}}, r_1) = \mathsf{SIG.RandPk}(\mathsf{pp_{SIG}}, \mathsf{pk_{SIG}}, r_2) \end{array} \middle| \begin{array}{r} \mathsf{pp_{SIG}} \leftarrow \mathsf{SIG.Setup}(1^\lambda) \\ (\mathsf{pk_{SIG}}, r_1, r_2) \leftarrow \mathcal{A}(\mathsf{pp_{SIG}}) \end{array}\right] .$$

Construction. In Fig. 21 we provide a modification of the Schnorr signature scheme [Sch91] that is randomizable. We briefly explain why this modification satisfies the security properties above.

- *Existential unforgeability under randomization (EUR).* Given an efficient adversary $\mathcal{A}$ that breaks EUR of randomizable Schnorr signatures, we construct an efficient adversary $\mathcal{A}'$ that breaks existential unforgeability of standard Schnorr signatures. In detail, $\mathcal{A}'$ forwards signature queries from $\mathcal{A}$ to its own signing oracle and returns the answers to $\mathcal{A}$ and then, when $\mathcal{A}$ outputs a tuple $(m^*, \sigma^*, r^*_{\mathsf{SIG}})$, $\mathcal{A}'$ outputs the tuple (m^*, σ) where σ is computed as follows. If σ^* is a valid signature for m^* under $\mathsf{pk}_{\mathsf{SIG}}$ then $\sigma := \sigma^*$. Otherwise, $\mathcal{A}'$ "undoes" the randomization of $\sigma^* = (s, e)$ by setting $\sigma := (s + e \cdot r^*_{\mathsf{SIG}}, e)$; thus if $\mathcal{A}$ outputs a forgery for a randomization of $\mathsf{pk}_{\mathsf{SIG}}$, $\mathcal{A}'$ translates it back into a forgery for $\mathsf{pk}_{\mathsf{SIG}}$. In sum, since standard Schnorr signatures are secure in the random oracle model assuming hardness of discrete logarithms [PS00], so is the randomizable variant under the same assumptions.
- *Unlinkability of public keys.* Public keys are unlinkable because $\mathsf{SIG.RandPk}$ multiplies the public key pk (which is a group element) by a random group element; the result is statistically independent of pk.
- *Unlinkability of signatures.* The only part of a Schnorr signature that depends on the public or secret key is the scalar s. Since $\mathsf{SIG.RandSig}$ adds a random shift to s, the result is statistically independent of the signature's original key pair.
- *Injective randomization.* Fixing all inputs but for r_{SIG}, $\mathsf{SIG.RandPk}$ is a permutation over $\mathbb{G}$. Hence, finding collisions over the randomness is not possible.

```
SIG.Setup($1^\lambda$) $\to \mathsf{pp}_{\mathsf{SIG}}$
1. Sample a group: $(\mathbb{G}, q, g) \leftarrow \mathsf{SampleGrp}(1^\lambda)$.
2. Sample cryptographic hash function $H$.
3. Output $\mathsf{pp}_{\mathsf{SIG}} := (\mathbb{G}, q, g, H)$.

SIG.Keygen($\mathsf{pp}_{\mathsf{SIG}}$) $\to (\mathsf{pk}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{SIG}})$
1. Parse $\mathsf{pp}_{\mathsf{SIG}}$ as $(\mathbb{G}, q, g, H)$.
2. Sample a scalar $x$ uniformly from $\mathbb{Z}_q$.
3. Output $(\mathsf{pk}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{SIG}}) := (g^x, x)$.

SIG.Verify($\mathsf{pp}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{SIG}}, m, \sigma$) $\to b$
1. Parse $\mathsf{pp}_{\mathsf{SIG}}$ as $(\mathbb{G}, q, g, H)$.
2. Parse $\sigma$ as $(s, e)$.
3. Set $r_v := g^s \mathsf{pk}_{\mathsf{SIG}}^e = g^{s+xe}$.
4. Set $e_v := H(r_v \| m)$.
5. Check if $e = e_v$.

SIG.Sign($\mathsf{pp}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{SIG}}, m$) $\to \sigma$
1. Parse $\mathsf{pp}_{\mathsf{SIG}}$ as $(\mathbb{G}, q, g, H)$.
2. Sample a scalar $k$ uniformly from $\mathbb{Z}_q$.
3. Set $r := g^k$ and $e := H(r \| m)$.
4. Set $s := k - xe$.
5. Output $\sigma := (s, e)$.

SIG.RandPk($\mathsf{pp}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{SIG}}, r_{\mathsf{SIG}}$) $\to \hat{\mathsf{pk}}_{\mathsf{SIG}}$
1. Parse $\mathsf{pp}_{\mathsf{SIG}}$ as $(\mathbb{G}, q, g, H)$.
2. Output $\hat{\mathsf{pk}}_{\mathsf{SIG}} := \mathsf{pk}_{\mathsf{SIG}} \cdot g^{r_{\mathsf{SIG}}}$.

SIG.RandSig($\mathsf{pp}_{\mathsf{SIG}}, \sigma, r_{\mathsf{SIG}}$) $\to \hat{\sigma}$
1. Parse $\mathsf{pp}_{\mathsf{SIG}}$ as $(\mathbb{G}, q, g, H)$.
2. Parse $\sigma$ as $(s, e)$.
3. Output $\hat{\sigma} := (s - e \cdot r_{\mathsf{SIG}}, e)$.
```

Figure 21: Construction of a randomizable signature scheme based on the Schnorr signature scheme [Sch91].

B.2 Construction of a delegable DPC scheme

Fig. 22 provides pseudocode that, together with the modified NP relation $\mathcal{R}_{\mathsf{e}}^{\mathsf{del}}$ given in Fig. 23, formalizes the high-level description of a delegable DPC scheme from Section 5.3. In both figures, we highlighted changes from the "plain" DPC scheme in Section 4.2. *The only step in* $\mathsf{DPC.Execute}$ *that must be performed by the delegator is Step 7a; all other steps can be performed by the worker without knowing the signature secret key.*

```
DPC.Setup
Input: security parameter $1^\lambda$
Output: public parameters $\mathsf{pp}$
1. Generate trapdoor commitment parameters:
   $\mathsf{pp}_{\mathsf{TCM}} \leftarrow \mathsf{TCM.Setup}(1^\lambda)$.
2. Generate CRH parameters: $\mathsf{pp}_{\mathsf{CRH}} \leftarrow \mathsf{CRH.Setup}(1^\lambda)$.
3. Generate signature parameters: $\mathsf{pp}_{\mathsf{SIG}} \leftarrow \mathsf{SIG.Setup}(1^\lambda)$.
4. Generate NIZK parameters for $\mathcal{R}_{\mathsf{e}}^{\mathsf{del}}$ (Fig. 23):
   $\mathsf{pp}_{\mathsf{e}} \leftarrow \mathsf{NIZK.Setup}(1^\lambda, \mathcal{R}_{\mathsf{e}}^{\mathsf{del}})$.
5. Output $\mathsf{pp} := (\mathsf{pp}_{\mathsf{TCM}}, \mathsf{pp}_{\mathsf{CRH}}, \mathsf{pp}_{\mathsf{SIG}}, \mathsf{pp}_{\mathsf{e}})$.
```

```
DPC.GenAddress
Input: public parameters $\mathsf{pp}$ and address metadata $\mathsf{meta}$
Output: address key pair $(\mathsf{apk}, \mathsf{ask})$
1. Generate authorization key pair:
   $(\mathsf{pk}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{SIG}}) \leftarrow \mathsf{SIG.Keygen}(\mathsf{pp}_{\mathsf{SIG}})$ .
2. Sample secret key $\mathsf{sk}_{\mathsf{PRF}}$ for pseudorandom function PRF.
3. Sample randomness $r_{\mathsf{pk}}$ for commitment scheme TCM.
4. Set address public key
   $\mathsf{apk} := \mathsf{TCM.Commit}(\mathsf{pp}_{\mathsf{TCM}}, \mathsf{pk}_{\mathsf{SIG}} \| \mathsf{sk}_{\mathsf{PRF}} \| \mathsf{meta};\ r_{\mathsf{pk}})$.
5. Set address secret key
   $\mathsf{ask} := (\mathsf{sk}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{PRF}}, \mathsf{meta}, r_{\mathsf{pk}})$.
6. Output $(\mathsf{apk}, \mathsf{ask})$.
```

```
DPC.Execute$^{\mathbf{L}}$
Input:
  • public parameters $\mathsf{pp}$
  • old { records $[\mathbf{r}_i]_1^m$
        { address secret keys $[\mathsf{ask}_i]_1^m$
  • auxiliary predicate input $\mathsf{aux}$
  • transaction memorandum $\mathsf{memo}$
  • new { address public keys $[\mathsf{apk}_j]_1^n$
        { record payloads $[\mathsf{payload}_j]_1^n$
        { record birth predicates $[\Phi_{\mathsf{b},j}]_1^n$
        { record death predicates $[\Phi_{\mathsf{d},j}]_1^n$
Output: new records $[\mathbf{r}_j]_1^n$ and transaction $\mathsf{tx}$
1. For each $i \in \{1, \dots, m\}$, process the $i$-th old record as follows:
   (a) Parse old record $\mathbf{r}_i$ as $\mathbf{r}_i = \begin{pmatrix} \text{address public key} & \mathsf{apk}_i & \text{payload} & \mathsf{payload}_i & \text{comm. rand.} & r_i \\ \text{serial number nonce} & \rho_i & \text{predicates} & (\Phi_{\mathsf{b},i}, \Phi_{\mathsf{d},i}) & \text{commitment} & \mathsf{cm}_i \end{pmatrix}$.
   (b) If $\mathsf{payload}_i.\mathsf{isDummy} = 1$, set ledger membership witness $\mathbb{w}_{\mathbf{L},i} := \bot$.
       If $\mathsf{payload}_i.\mathsf{isDummy} = 0$, compute ledger membership witness for commitment: $\mathbb{w}_{\mathbf{L},i} \leftarrow \mathbf{L}.\mathsf{Prove}(\mathsf{cm}_i)$.
   (c) Parse address secret key $\mathsf{ask}_i$ as $(\mathsf{sk}_{\mathsf{SIG},i}, \mathsf{sk}_{\mathsf{PRF},i}, \mathsf{meta}_i, r_{\mathsf{pk},i})$ and derive $\mathsf{pk}_{\mathsf{SIG},i}$ from $\mathsf{sk}_{\mathsf{SIG},i}$.
   (d) Compute signature randomness: $r_{\mathsf{SIG},i} \leftarrow \mathsf{PRF}_{\mathsf{sk}_{\mathsf{PRF},i}}(\rho_i)$.
   (e) Compute serial number: $\mathsf{sn}_i \leftarrow \mathsf{SIG.RandPk}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{SIG},i}, r_{\mathsf{SIG},i})$.
2. For each $j \in \{1, \dots, n\}$, construct the $j$-th new record as follows:
   (a) Compute serial number nonce: $\rho_j := \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, j \| \mathsf{sn}_1 \| \dots \| \mathsf{sn}_m)$.
   (b) Construct new record: $\mathbf{r}_j \leftarrow \mathsf{DPC.ConstructRecord}(\mathsf{pp}, \mathsf{apk}_j, \mathsf{payload}_j, \Phi_{\mathsf{b},j}, \Phi_{\mathsf{d},j}, \rho_j)$.
3. Retrieve current ledger digest: $\mathsf{st}_{\mathbf{L}} \leftarrow \mathbf{L}.\mathsf{Digest}$.
4. Construct instance for relation $\mathcal{R}_{\mathsf{e}}^{\mathsf{del}}$: $\mathbb{x}_{\mathsf{e}} := (\mathsf{st}_{\mathbf{L}}, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo})$.
5. Construct witness for relation $\mathcal{R}_{\mathsf{e}}^{\mathsf{del}}$: $\mathbb{w}_{\mathsf{e}} := ([\mathbf{r}_i]_1^m, [\mathbb{w}_{\mathbf{L},i}]_1^m, [\mathsf{sk}_{\mathsf{PRF},i}]_1^m, [\mathsf{pk}_{\mathsf{SIG},i}]_1^m, [\mathsf{meta}_i]_1^m, [r_{\mathsf{pk},i}]_1^m, [\mathbf{r}_j]_1^n, \mathsf{aux})$.
6. Generate proof for relation $\mathcal{R}_{\mathsf{e}}^{\mathsf{del}}$: $\pi_{\mathsf{e}} \leftarrow \mathsf{NIZK.Prove}(\mathsf{pp}_{\mathsf{e}}, \mathbb{x}_{\mathsf{e}}, \mathbb{w}_{\mathsf{e}})$.
7. For each $i \in \{1, \dots, m\}$:
   (a) Sign message: $\sigma_i \leftarrow \mathsf{SIG.Sign}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{sk}_{\mathsf{SIG},i}, \mathbb{x}_{\mathsf{e}} \| \pi_{\mathsf{e}})$.
   (b) Randomize signature: $\hat{\sigma}_i \leftarrow \mathsf{SIG.RandSig}(\mathsf{pp}_{\mathsf{SIG}}, \sigma_i, r_{\mathsf{SIG},i})$.
8. Construct transaction: $\mathsf{tx} := ([\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo}, \star)$, where $\star := (\mathsf{st}_{\mathbf{L}}, \pi_{\mathsf{e}}, [\hat{\sigma}_i]_1^m)$.
9. Output $([\mathbf{r}_j]_1^n, \mathsf{tx})$.
```

```
DPC.Verify$^{\mathbf{L}}$
Input: public parameters $\mathsf{pp}$ and transaction $\mathsf{tx}$
Output: decision bit $b$
1. Parse $\mathsf{tx}$ as $([\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo}, \star)$ and $\star$ as $(\mathsf{st}_{\mathbf{L}}, \pi_{\mathsf{e}}, [\hat{\sigma}_i]_1^m)$.
2. Check that there are no duplicate serial numbers
   (a) within the transaction tx: $\mathsf{sn}_i \neq \mathsf{sn}_j$ for every distinct $i, j \in \{1, \dots, m\}$;
   (b) on the ledger: $\mathbf{L}.\mathsf{Contains}(\mathsf{sn}_i) = 0$ for every $i \in \{1, \dots, m\}$.
3. Check that the ledger state is valid: $\mathbf{L}.\mathsf{ValidateDigest}(\mathsf{st}_{\mathbf{L}}) = 1$.
4. Construct instance for the relation $\mathcal{R}_{\mathsf{e}}^{\mathsf{del}}$: $\mathbb{x}_{\mathsf{e}} := (\mathsf{st}_{\mathbf{L}}, [\mathsf{sn}_i]_1^m, [\mathsf{cm}_j]_1^n, \mathsf{memo})$.
5. Check proof for the relation $\mathcal{R}_{\mathsf{e}}^{\mathsf{del}}$: $\mathsf{NIZK.Verify}(\mathsf{pp}_{\mathsf{e}}, \mathbb{x}_{\mathsf{e}}, \pi_{\mathsf{e}}) = 1$.
6. For every $i \in \{1, \dots, m\}$, check that signature verifies: $\mathsf{SIG.Verify}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{sn}_i, \mathbb{x}_{\mathsf{e}} \| \pi_{\mathsf{e}}, \hat{\sigma}_i) = 1$.
```

Figure 22: Construction of a delegable DPC scheme. Highlights denote differences from Figure 8.

The NP relation $\mathcal{R}_{\mathsf{e}}^{\mathsf{del}}$ has instances $\mathbb{x}_{\mathsf{e}}$ and witnesses $\mathbb{w}_{\mathsf{e}}$ of the following form.

$$\mathbb{x}_{\mathsf{e}} = \begin{pmatrix} \text{ledger digest} & \mathsf{st}_{\mathbf{L}} \\ \text{old record serial numbers} & [\mathsf{sn}_i]_1^m \\ \text{new record commitments} & [\mathsf{cm}_j]_1^n \\ \text{transaction memorandum} & \mathsf{memo} \end{pmatrix} \quad \text{and} \quad \mathbb{w}_{\mathsf{e}} = \begin{pmatrix} \text{old records} & [\mathbf{r}_i]_1^m \\ \text{old record membership witnesses} & [\mathbb{w}_{\mathbf{L},i}]_1^m \\ \text{old record authorization public keys} & [\mathsf{sk}_{\mathsf{PRF},i}]_1^m \\ \text{old record serial number secret keys} & [\mathsf{pk}_{\mathsf{SIG},i}]_1^m \\ \text{old record address metadata} & [\mathsf{meta}_i]_1^m \\ \text{old record address randomness} & [r_{\mathsf{pk},i}]_1^m \\ \text{new records} & [\mathbf{r}_j]_1^n \\ \text{auxiliary predicate input} & \mathsf{aux} \end{pmatrix}$$

where

- for each $i \in \{1,\ldots,m\}$, $\mathbf{r}_i = (\mathsf{apk}_i, \mathsf{payload}_i, \Phi_{\mathsf{b},i}, \Phi_{\mathsf{d},i}, \rho_i, r_i, \mathsf{cm}_i)$;
- for each $j \in \{1,\ldots,n\}$, $\mathbf{r}_j = (\mathsf{apk}_j, \mathsf{payload}_j, \Phi_{\mathsf{b},j}, \Phi_{\mathsf{d},j}, \rho_j, r_j, \mathsf{cm}_j)$.

Define the local data $\mathsf{ldata} := \begin{pmatrix} [\mathsf{cm}_i]_1^m & [\mathsf{apk}_i]_1^m & [\mathsf{payload}_i]_1^m & [\Phi_{\mathsf{d},i}]_1^m & [\Phi_{\mathsf{b},i}]_1^m & [\mathsf{meta}_i]_1^m & [\mathsf{sn}_i]_1^m \\ [\mathsf{cm}_j]_1^n & [\mathsf{apk}_j]_1^n & [\mathsf{payload}_j]_1^n & [\Phi_{\mathsf{d},j}]_1^n & [\Phi_{\mathsf{b},j}]_1^n & \mathsf{memo} & \mathsf{aux} \end{pmatrix}$.

A witness $\mathbb{w}_{\mathsf{e}}$ is valid for an instance $\mathbb{x}_{\mathsf{e}}$ if the following conditions hold:

1. For each $i \in \{i,\ldots,m\}$:
   - If $\mathbf{r}_i$ is not dummy, $\mathbb{w}_{\mathbf{L},i}$ proves that the commitment cm_i is in a ledger with digest $\mathsf{st}_{\mathbf{L}}$: $\mathbf{L}.\mathsf{Verify}(\mathsf{st}_{\mathbf{L}}, \mathsf{cm}_i, \mathbb{w}_{\mathbf{L},i}) = 1$.
   - The address public key apk_i matches the authorization public key $\mathsf{pk}_{\mathsf{SIG},i}$ and the serial number secret key $\mathsf{sk}_{\mathsf{PRF},i}$: $\mathsf{apk}_i = \mathsf{TCM.Commit}(\mathsf{pp}_{\mathsf{TCM}}, \mathsf{pk}_{\mathsf{SIG},i} \| \mathsf{sk}_{\mathsf{PRF},i} \| \mathsf{meta}_i; r_{\mathsf{pk},i})$.
   - The serial number sn_i is valid: $r_{\mathsf{SIG},i} = \mathsf{PRF}_{\mathsf{sk}_{\mathsf{PRF},i}}(\rho_i)$ and $\mathsf{sn}_i = \mathsf{SIG.RandPk}(\mathsf{pp}_{\mathsf{SIG}}, \mathsf{pk}_{\mathsf{SIG},i}, r_{\mathsf{SIG},i})$.
   - The old record commitment cm_i is valid: $\mathsf{cm}_i = \mathsf{TCM.Commit}(\mathsf{pp}_{\mathsf{TCM}}, \mathsf{apk}_i \| \mathsf{payload}_i \| \Phi_{\mathsf{b},i} \| \Phi_{\mathsf{d},i} \| \rho_i;\ r_i)$.
   - The death predicate $\Phi_{\mathsf{d},i}$ is satisfied by the local data: $\Phi_{\mathsf{d},i}(i \| \mathsf{ldata}) = 1$.
2. For each $j \in \{1,\ldots,n\}$:
   - The serial number nonce ρ_j is computed correctly: $\rho_j = \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, j \| \mathsf{sn}_1 \| \ldots \| \mathsf{sn}_m)$.
   - The new record commitment cm_j is valid: $\mathsf{cm}_j = \mathsf{TCM.Commit}(\mathsf{pp}_{\mathsf{TCM}}, \mathsf{apk}_j \| \mathsf{payload}_j \| \Phi_{\mathsf{b},j} \| \Phi_{\mathsf{d},j} \| \rho_j;\ r_j)$.
   - The birth predicate $\Phi_{\mathsf{b},j}$ is satisfied by the local data: $\Phi_{\mathsf{b},j}(j \| \mathsf{ldata}) = 1$.

Figure 23: The NP relation $\mathcal{R}_{\mathsf{e}}^{\mathsf{del}}$. Highlights denote differences from Figure 9.

C Extensions in functionality and in security

We summarize some natural extensions of our DPC construction that give richer functionality, as well as methods to prove security notions beyond standalone non-adaptive security.

On-ledger encryption. A user can store an encryption public key in the metadata of one of its addresses. Others can then use this public key to encrypt information about records created for the user, and store the resulting ciphertext in the transaction's memorandum. This method, used for example in Zerocash [BCG+14], gives users the option to not use other out-of-band secure communication channels.

Ledger position. In some applications it may be useful to know the unique ledger position of a record, i.e., to have this information be part of the local data ldata given as input to predicates. For example, one can use a record's ledger position to implement a "time lock" that prevents the record's consumption until a pre-specified amount of time has passed since the record's creation. However, the ledger interface we described in Section 3.1 does not expose this functionality: $\mathbf{L}.\mathsf{Prove}$ only returns a proof that a transaction (or a subcomponent thereof) appears on the ledger, and *not its position*. One can augment $\mathbf{L}.\mathsf{Prove}$ to instead output the transaction's ledger position $\mathsf{pos}_{\mathbf{L}}$, and a proof that $\mathsf{pos}_{\mathbf{L}}$ is the transaction's position on the ledger. Our instantiation of the ledger with a Merkle tree supports this augmentation inherently: the path to the transaction in the Merkle tree is also its position the tree.

Composable security. The security definition in Section 3.3 is a restriction of UC security definitions to a single execution at any given time. We can avoid this restriction and prove our construction UC-secure by replacing our simulation-extractable NIZKs with UC-secure NIZKs. The remainder of the proof would go through unchanged, and this would achieve composition of multiple protocol instances.

Adaptive security. We can prove adaptive security, with a minor modification to our protocol in Section 4. The barrier to proving security against adaptive corruptions (even in a standalone setting) is a lack of forward-secure privacy. Namely, when the adversary corrupts a party $\mathcal{P}$, it gets access to $\mathcal{P}$'s state, which includes contents of records held by $\mathcal{P}$ and address secret keys belonging to $\mathcal{P}$. The adversary can then use this information to break unlinkability of $\mathcal{P}$'s transactions by deriving the serial numbers of consumed records and matching these against those present on the ledger.

In the proof, this problem is reflected in how the simulator $\mathcal{S}$ handles serial numbers in honest party transactions (see Appendix A.2). For honest party transactions, serial numbers are sampled uniformly at random via $\mathsf{SampleSn}$. When the environment $\mathcal{E}$ corrupts an honest party, it can attempt to carry out the aforementioned linking attack by computing serial numbers via the PRF. Since serial numbers already published in transactions were derived randomly, they would not match the output of the PRF, allowing $\mathcal{E}$ to distinguish the ideal world from the real world.

We address this issue as follows. First, we work in the secure-erasure model and ensure that honest parties delete (a) all records output from $\mathsf{Execute}$ (after sending their contents to the intended recipients), and (b) all records that have been consumed. Hence, at the time a party is corrupted, the state revealed to the adversary does not contain secrets of past records, so the adversary cannot derive those records' serial numbers. However this by itself is not enough. Consider the following scenario: the adversary corrupts an honest user and learns her secret key. For every transaction in the ledger, it computes the serial number nonces of the output records from the serial numbers of the input records. The adversary can then use these nonces along with the secret key to derive candidate serial numbers for the output records. If these candidate serial numbers appear on the ledger, then the adversary learns that the record has been consumed.

To prevent this, we randomize the serial number nonces of all records output by $\mathsf{Execute}$ by deriving them as $\rho_j := \mathsf{CRH}(j \| r_{\rho,j} \| \mathsf{sn}_1 \| \cdots \| \mathsf{sn}_m)$ for some randomness $r_{\rho,j}$ that is deleted after invoking $\mathsf{Execute}$. This randomization ensures that the serial number nonce of an output record cannot be derived deterministically from the (publicly visible) serial numbers of the input records.

The above measures, however, are still insufficient: the adversary still knows the secrets of records that a corrupted party sent to an honest party. After corrupting this honest party, the adversary can learn its address secret key and therefore derive the serial number of those records. To overcome this obstacle, one can replace the PRF with a *programmable PRF* [PS18], for which the owner of the secret key can "program" the PRF to output pre-determined values on specific inputs: for all polynomial-sized sets $S = \{(x_i, y_i)\}_i$, the owner of a PRF secret key sk can derive a second key sk_S such that $\mathsf{PRF}_{\mathsf{sk}_s}(x_i) = y_i$ for each $(x_i, y_i) \in S$, while $\mathsf{PRF}_{\mathsf{sk}_S}(x) = \mathsf{PRF}_{\mathsf{sk}}(x)$ for other inputs x. This fixes the foregoing issue because $\mathcal{S}$ can now give $\mathcal{E}$ a programmed PRF secret key for the set $S = \{(\rho_i, \mathsf{sn}_i)\}_i$, where ρ_i is the serial number nonce of the i-th record received from a corrupted party.

Acknowledgments

This work was supported in part by: a Google Faculty Award; the National Science Foundation under awards CNS-1653110 and CNS-1801479; the UC Berkeley Center for Long-Term Cybersecurity; and donations from the Interchain Foundation and Qtum.

References

[ADMM14a] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Fair two-party computations via Bitcoin deposits. In *Proceedings of the BITCOIN workshop at the 18th International Conference on Financial Cryptography and Data Security*, FC '14, pages 105–121, 2014.

[ADMM14b] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on Bitcoin. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, SP '14, pages 443–458, 2014.

[AFK+12] Diego F. Aranha, Laura Fuentes-Castañeda, Edward Knapp, Alfred Menezes, and Francisco Rodríguez-Henríquez. Implementing pairings at the 192-bit security level. In *Proceedings of the 5th International Conference on Pairing-Based Cryptography*, Pairing '12, pages 177–195, 2012.

[AKR+13] Elli Androulaki, Ghassan Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. Evaluating user privacy in Bitcoin. In *Proceedings of the 17th International Conference on Financial Cryptography and Data Security*, FC '13, pages 34–51, 2013.

[ANWW13] Jean-Phillipe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS '13, pages 119–135, 2013.

[BCCT13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *Proceedings of the 45th ACM Symposium on the Theory of Computing*, STOC '13, pages 111–120, 2013.

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: verifying program executions succinctly and in zero knowledge. In *Proceedings of the 33rd Annual International Cryptology Conference*, CRYPTO '13, pages 90–108, 2013.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 459–474, 2014. Full version available at `http://eprint.iacr.org/2014/349`.

[BCG+15] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, SP '15, pages 287–304, 2015.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *Proceedings of the 23rd USENIX Security Symposium*, USENIX '14, pages 781–796, 2014.

[BCTV17] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica*, 79(4):1102–1160, 2017.

[BGG17] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the pinocchio zk-snark. Cryptology ePrint Archive, Report 2017/602, 2017. `http://eprint.iacr.org/2017/602`.

[Bit15] Bitcoin. Some miners generating invalid blocks. `https://bitcoin.org/en/alert/2015-07-04-spv-mining`, 2015.

[BKM17] Iddo Bentov, Ranjit Kumaresan, and Andrew Miller. Instantaneous decentralized poker. In *Proceedings of the 23rd Annual International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '17, pages 410–440, 2017.

[BLS02] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *Proceedings of the 3rd International Conference on Security in Communication Networks*, SCN '02, pages 257–267, 2002.

[Bow17a] Sean Bowe. Bellman, 2017. `https://github.com/zkcrypto/bellman`.

[Bow17b] Sean Bowe. Pairing, 2017. `https://github.com/zkcrypto/pairing`.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science*, FOCS '01, pages 136–145, 2001.

[CCW18] Alessandro Chiesa, Lynn Chua, and Matthew Weidner. On cycles of pairing-friendly elliptic curves. arXiv math.NT/1803.02067, 2018. `https://arxiv.org/abs/1803.02067`.

[CGL$^+$17] Alessandro Chiesa, Matthew Green, Jingcheng Liu, Peihan Miao, Ian Miers, and Pratyush Mishra. Decentralized anonymous micropayments. In *Proceedings of the 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT '17, pages 609–642, 2017.

[Cha14] Chainalysis. Chainalysis inc. `https://chainalysis.com/`, 2014.

[CLN11] Craig Costello, Kristin E. Lauter, and Michael Naehrig. Attractive subfamilies of BLS curves for implementing high-security pairings. In *Proceedings of the 12th International Conference on Cryptology in India*, INDOCRYPT '11, pages 320–342, 2011.

[Cos12] Craig Costello. Particularly friendly members of family trees. Cryptology ePrint Archive, Report 2012/072, 2012.

[CRR11] Ran Canetti, Ben Riva, and Guy N. Rothblum. Practical delegation of computation using multiple servers. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS '11, pages 445–454, 2011.

[CRR13] Ran Canetti, Ben Riva, and Guy N. Rothblum. Refereed delegation of computation. *Information and Computation*, 226:16–36, 2013.

[CZK$^+$18] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nichola Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution. arXiv cs.CR/1804.05141, 2018. `https://arxiv.org/abs/1804.05141`.

[DDO$^+$01] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. Robust non-interactive zero knowledge. In *Proceedings of the 21st Annual International Cryptology Conference*, CRYPTO '01, pages 566–598, 2001.

[DF91] Yvo Desmedt and Yair Frankel. Shared generation of authenticators and signatures (extended abstract). In *Proceedings of the 11th Annual International Cryptology Conference*, CRYPTO '91, pages 457–469, 1991.

[Dod07] Yevgeniy Dodis. Lecture notes: Exposure-resilient cryptography, 2007. `https://www.cs.nyu.edu/courses/spring07/G22.3033-013/`.

[DSC$^+$15] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2R: enabling stronger privacy in MapReduce computation. In *Proceedings of the 24th USENIX Security Symposium*, Security '15, pages 447–462, 2015.

[Ell13] Elliptic. Elliptic enterprises limited. https://www.elliptic.co/, 2013.

[Eth16] Ethereum. I thikn the attacker is this miner - today he made over $50k. https://www.reddit.com/r/ethereum/comments/55xh2w/i_thikn_the_attacker_is_this_miner_today_he_made/, 2016.

[Eth18] Etherscan. The ethereum block explorer, 2018. https://etherscan.io/tokens.

[FK97] Uriel Feige and Joe Kilian. Making games short. In *Proceedings of the 29th ACM Symposium on the Theory of Computing*, STOC '97, pages 506–516, 1997.

[FKM$^+$16] Nils Fleischhacker, Johannes Krupp, Giulio Malavolta, Jonas Schneider, Dominique Schröder, and Mark Simkin. Efficient unlinkable sanitizable signatures from signatures with re-randomizable keys. In *Proceedings of the 19th International Conference on Practice and Theory in Public-Key Cryptography*, PKC '16, pages 301–330, 2016.

[FST10] David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of cryptology*, 23(2):224–280, 2010.

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. Cryptology ePrint Archive, Report 2017/540, 2017. http://eprint.iacr.org/2017/540.

[Gro06] Jens Groth. Simulation-sound NIZK proofs for a practical language and constant size group signatures. In *Proceedings of the 12th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '06, pages 444–459, 2006.

[HBHW18] Daira Hopwood, Sean Bowe, Tailor Hornby, and Nathan Wilcox. Zcash protocol specification, 2018. URL: https://github.com/zcash/zips/blob/master/protocol/protocol.pdf.

[IKOS06] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography from anonymity. In *Proceedings of the 47th Annual Symposium on Foundations of Computer Science*, FOCS '06, pages 239–248, 2006.

[JSST16] Sanjay Jain, Prateek Saxena, Frank Stephan, and Jason Teutsch. How to verify computation with a rational network. arXiv cs.GT/1606.05917, 2016. https://arxiv.org/abs/1606.05917.

[KB16] Ranjit Kumaresan and Iddo Bentov. Amortizing secure computation with penalties. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, CCS '16, pages 418–429, 2016.

[KGC$^+$17] Harry Kalodner, Steven Goldfeder, Alishah Chator, Malte Möser, and Arvind Narayanan. BlockSci: design and applications of a blockchain analysis platform. arXiv cs.CR/1709.02489, 2017. https://arxiv.org/abs/1709.02489.

[KGC$^+$18] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *Proceedings of the 27th USENIX Security Symposium*, Security '18, pages 1353–1370, 2018.

[KMB15] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use Bitcoin to play decentralized poker. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS '15, pages 195–206, 2015.

[KMS$^+$16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '16, pages 839–858, 2016.

[LTKS15] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS '15, pages 706–719, 2015.

[MAB$^+$13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the Second Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, page 10, 2013.

[Mer87] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the 7th Conference on the Theory and Applications of Cryptographic Techniques*, CRYPTO '87, pages 369–378, 1987.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed e-cash from Bitcoin. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 397–411, 2013.

[MPJ$^+$13] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M. Voelker, and Stefan Savage. A fistful of Bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 Internet Measurement Conference*, IMC '13, pages 127–140, 2013.

[MRK03] Silvio Micali, Michael O. Rabin, and Joe Kilian. Zero-knowledge sets. In *Proceedings of the 44th Annual Symposium on Foundations of Computer Science*, FOCS '03, pages 80–91, 2003.

[Nak09] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2009. URL: `http://www.bitcoin.org/bitcoin.pdf`.

[NKDM03] Antonio Nicolosi, Maxwell N. Krohn, Yevgeniy Dodis, and David Mazières. Proactive two-party signatures for user authentication. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS '03, 2003.

[PB17] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. `https://plasma.io/`, 2017.

[PS00] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *J. Cryptology*, 13(3):361–396, 2000.

[PS18] Chris Peikert and Sina Shiehian. Privately constraining and programming PRFs, the LWE way. In *Proceedings of the 21st International Conference on Practice and Theory in Public-Key Cryptography*, PKC '18, pages 675–701, 2018.

[Rei16] Christian Reiwießner. From smart contracts to courts with not so smart judges. `https://blog.ethereum.org/2016/02/17/smart-contracts-courts-not-smart-judges/`, 2016.

[RH11] Fergal Reid and Martin Harrigan. An analysis of anonymity in the Bitcoin system. In *Proceedings of the 3rd IEEE International Conference on Privacy, Security, Risk and Trust (PASSAT), and the 3rd IEEE International Conference on Social Computing (SocialCom)*, SocialCom/PASSAT '11, pages 1318–1326, 2011.

[RS13] Dorit Ron and Adi Shamir. Quantitative analysis of the full Bitcoin transaction graph. In *Proceedings of the 17th International Conference on Financial Cryptography and Data Security*, FC '13, pages 6–24, 2013.

[Sah99] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 543–553, 1999.

[SCF$^+$15] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, SP '15, pages 38–54, 2015.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[SJ99] Claus-Peter Schnorr and Markus Jakobsson. Security of discrete log cryptosystems in the random oracle and the generic model. In *The Mathematics of Public-Key Cryptography*, MPKC '99, 1999.

[SMZ14] Michele Spagnuolo, Federico Maggi, and Stefano Zanero. BitIodine: Extracting intelligence from the Bitcoin network. In *Proceedings of the 18th International Conference on Financial Cryptography and Data Security*, FC '14, pages 457–468, 2014.

[SS01] Douglas R. Stinson Stinson and Reto Strobl. Provably secure distributed Schnorr signatures and a (t, n) threshold scheme for implicit certificates. In *Proceedings of the 5th Australasian Conference on Information Security and Privacy*, ACISP '01, pages 417–434, 2001.

[TR17] Jason Teutsch and Christian Reiwießner. A scalable verification solution for blockchains. `https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf`, 2017.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *Proceedings of the 5th Theory of Cryptography Conference*, TCC '08, pages 1–18, 2008.

[VB15] Fabian Vogelsteller and Vitalik Buterin. ERC-20 token standard, 2015. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`.

[Woo17] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2017. `http://yellowpaper.io`.

[WSR$^+$15] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, NDSS '15, 2015.

[ZCa15] ZCash Company, 2015. `https://z.cash/`.

[ZCa16] ZCash parameter generation. `https://z.cash/technology/paramgen.html`, 2016. Accessed: 2017-09-28.

[ZDB$^+$17] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, pages 283–298, 2017.