Zexe: Enabling Decentralized Private Computation

Sean Bowe*, Alessandro Chiesa†, Matthew Green‡, Ian Miers§, Pratyush Mishra†, Howard Wu†
*Zcash, sean@z.cash
†UC Berkeley, {alexch,pratyush,howardwu}@berkeley.edu
‡Johns Hopkins University, mgreen@cs.jhu.edu
§Cornell Tech, imiers@cs.cornell.edu

***Abstract*—Ledger-based systems that support 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.**

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 in constant time* by anyone, regardless of the offline computation.

The core of Zexe is a construction for a new cryptographic primitive that we introduce, *decentralized private computation* (DPC) schemes. In order to achieve an efficient implementation of our construction, 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 $1\,\mathrm{min}$ plus a time that grows with the offline computation.

We demonstrate how to use Zexe to realize privacy-preserving analogues of popular applications: private user-defined assets and private decentralized exchanges for these assets.

I. Introduction

Distributed ledgers are a mechanism for maintaining 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, thus enabling multiple parties to collaborate with minimal trust, as 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*.

A 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 necessarily 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 *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.

A scalability problem. Public auditability in the aforementioned systems (and many others) is achieved via direct verification of state transitions that re-executes the associated computation. This creates the following scalability issues. First, note that in a network consisting of devices with heterogeneous computing power, requiring every node to re-execute transactions makes the weakest node a bottleneck, and this effect persists even when the underlying ledger is "perfect", that is, it confirms every valid transaction immediately. To counteract this and 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. However, such mechanisms can make it unprofitable to validate legitimate but expensive transactions, a problem known as the "Verifier's

[1] Even just revealing *addresses* in transactions can reveal much information about the flow of money [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.

DOI 10.1109/SP40000.2020.00050

Dilemma" [LTK+15]. 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 I-B below.

A. 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 indistinguishable, and are hence equally cheap to validate, there is no "Verifier's Dilemma", nor a need for mechanisms like Ethereum's 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 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 for performing computations on a ledger called *decentralized private computation* (DPC). Informally, a DPC scheme supports a simple, yet expressive, programming model in which units of data, which we call *records*, are bound to scripts (arbitrary programs) that specify the conditions under which a record can be created and consumed (this model is similar to the UTXO model; see Remark III.3). 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 captured via an ideal functionality, which our protocol provably achieves.

Applications. To illustrate the expressivity of the RNK, we show how to use DPC schemes to construct privacy-preserving analogues of popular applications such as private user-defined assets and private decentralized or non-custodial exchanges (DEXs). Our privacy guarantees in particular protect against vulnerabilities of current DEX designs such as front-running [BDJ+17; BBD+17; EMC19; DGK+19]. Moreover, we sketch how to use DPC to construct a privacy-preserving smart contract system. See Sections III-A and V for details.

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 a minute 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. 6), 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).

A perspective on costs. ZEXE is not a lightweight construction, but achieves, in our opinion, tolerable efficiency for the ambitious goals it sets out to achieve: *data and function privacy, and succinctness, with 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.

The primary cost in our system is, unsurprisingly, the cost of generating the cryptographic proofs that are included in transactions. We have managed to keep this cost to roughly a minute plus a cost that grows with the offline computation. For the applications mentioned above, these additional costs are negligible. Our system thus supports applications of real-world interest today (e.g., private DEXs) with reasonable costs.

B. Related work

Avoiding naive re-execution. A number of proposals for improving the scalability of smart contract systems, such as

[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.

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; JSS+16; 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 any private inputs).

Private payments. Zerocash [BCG+14], building on earlier work [MGG+13], showed how to use distributed ledgers to achieve payment systems with strong privacy guarantees. The Zerocash protocol, with some modifications, is now commercially deployed in several cryptocurrencies, including Zcash [Zcaa]. Solidus [CZJ+17] enables customers of financial institutions (such as banks) to transfer funds to one another in a manner that ensures that only the banks of the sender and receiver learn the details of the transfer; all other parties (all other customers and banks) only learn that a transfer occurred, and nothing else. zkLedger [NVV18] enables anonymous payments between a small number of distinguished parties via the use of homomorphic commitments and Schnorr proofs. None of these protocols support scripts (small programs that dictate how funds can be spent), let alone arbitrary state transitions as in Zexe.

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. Hawk's privacy guarantee protects the private inputs used in a computation, but does not hide *which* computation was performed. 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 then additionally hide *which* computation was carried out offline.

Zether [BAZ+19] is a system that enables *publicly known* smart contracts to reason about homomorphic commitments in zero knowledge, and in particular enables these to transact in a manner that hides transaction amounts; it does not hide the identities of parties involved in the transaction, beyond a small anonymity set. Furthermore, the cost of verifying a transaction scales linearly with the size of the anonymity set, whereas in Zexe this cost scales logarithmically with the size of anonymity set.

Succinct blockchains. Coda [MS18] uses arbitrary-depth recursive composition of SNARKs to enable blockchain nodes to verify the current blockchain state quickly. In contrast, Zexe uses depth-2 recursive composition to ensure that all blockchain transactions are equally cheap to verify (and are moreover indistinguishable from each other), regardless of the cost of the offline computation. In this respect, Coda and Zexe address orthogonal scalability concerns.

MPC with ledgers. Several works [ADM+14b; ADM+14a; KMB15; KB16; BKM17; RCGJ+17] 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. Kaptchuk et al. [KGM19] and Ekiden [CZK+18] combine ledgers with 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). Moreover, the techniques above depend on secure *remote attestation* capabilities, which have recently been broken for systems like SGX [VBMW+19].

II. Technical challenges

We now describe the key technical challenges that arise when trying to design a ledger-based system which achieves the goals of this paper, namely enabling arbitrary offline computations while simultaneously providing *privacy* and *succinctness*.

Most of the challenges we face revolve around achieving privacy. Indeed, if privacy is not required, there is a straightforward folklore approach that provides succinctness and low verification cost: 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 simply verify the cryptographic proof, and do not have to re-execute the computation. Even this limited approach rules out a number of cryptographic directions, such as the use of Bulletproofs [BCC+16; BBB+18] (which have verification time linear in the circuit complexity), but can be accomplished using a number of efficient SNARK techniques [GGP+13; BCT+14; BCS16; BCT+17]. In light of this, we shall first discuss the challenges that arise in achieving privacy.

A. 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 publicly known.

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.

B. Difficulties with achieving privacy for user-defined functions

We want to enable users to execute functions of their choice concurrently on the same ledger without seeking permission from anyone. That is, when preparing a transaction, a user should be able to pick *any* function Φ of their choice for creating new records by consuming some old records. If function privacy is not a concern, then this is easy: just attach to the transaction a zero-knowledge proof that Φ was correctly evaluated offline. However, because this approach reveals Φ, we cannot use it because function privacy is a goal for us.

An approach that *does* achieve function privacy would be to modify the sketch in Section II-A 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. A key challenge in this setting is that malicious 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, a verifier would not know what functions were used to create those records. For a concrete 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 $\Phi_\$$ operating on prior records; a malicious user could have instead used a different function to create that record, for example, one that illegally "mints" records that appear valid to $\Phi_\$$, and thus enables arbitrary inflation of the currency. More generally, the lack of any enforced rules about how user-defined functions can interact precludes productive cooperation between users that are mutually distrustful. We stress that this challenge arises specifically due to the requirement that functions be private: 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 "good" functions.

One way to address the foregoing problem is to augment records with a new attribute that identifies the function that "created" the record, and then impose the restriction that in a valid transaction only records created by the same function may participate. This new attribute is contained within a hiding commitment and thus is never revealed publicly on the ledger (just like a record's payload); 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 II-A, 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: functions cannot communicate. The limitation of the above technique is that it forbids any "inter-process communication" between different functions, and so one cannot realize even simple functionalities like transferring value between different currencies on the same ledger. It also rules out more complex smart contract systems, as communication between contracts is a key part of such systems. It is thus clear that this crude "time sharing" of the ledger is too limiting.

III. Our system design

The approaches in Section II-B lie at opposite extremes: unrestricted inter-process interaction prevents the secure construction of even basic applications such as a single currency,

while complete process segregation limits the ability to construct complex applications that interact with with each other.

Balancing these extremes requires a shared execution environment: one can think of this as an *operating system* for a shared ledger. This operating system 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, and enables programming applications in the UTXO model (see Remark III.3 for details on privacy in the UTXO model). Section III-A describes this shared execution environment, which we call the "records nano-kernel". Section III-B then shows how to realise this nano-kernel via a novel cryptographic primitive, *decentralized private computation schemes*.

A. The records nano-kernel: a minimalist shared execution environment

As stated above, our setting calls for a minimalist shared execution environment, or "nano-kernel", that enables users to manage records containing data by programming two boolean functions (or predicates) associated with each record. These predicates control the two defining moments in a record's life, namely creation (or "birth") and consumption (or "death"), and are hence called the record's *birth* and *death* predicates. A user can create and consume records in a transaction by satisfying the predicates of those records. In more detail,

> **The records nano-kernel (RNK)** is an execution environment that operates over units of data called *records*. A record contains a *data payload*, a *birth predicate* Φ_{b}, and a *death predicate* Φ_{d}. Records are created and consumed by *valid transactions*. These are transactions where the death predicates of all consumed records and the birth predicates of all created records are simultaneously satisfied when given as input the transaction's *local data* (see Fig. 4), 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.

The foregoing definition enables predicates to see the contents of the entire transaction and hence *to individually decide if the local data is valid according to its own logic*. This in turn enables predicates to communicate with each other in a secure manner without interference from malicious predicates. In more detail, a record $\mathbf{r}$ can protect itself from other records that contain "bad" birth or death predicates the $\mathbf{r}$'s predicates could refuse to accept when they detect (from reading the local data) that they are part of a transaction containing 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, thus providing the flexibility that we seek.

We briefly illustrate this via an example, *user-defined assets*, whereby one can use birth predicates to define and transact with their own assets, and also use death predicates to enforce custom access control policies over these assets.

Example III.1 (user-defined assets)**.** Consider records whose payloads encode an asset identifier id, the initial asset supply $\mathbb{v}$, and a value v. Fix the birth predicate in all such records to be a *mint-or-conserve* function MoC that is responsible for creating the initial supply of a new asset, and then subsequently conserving the value of the asset across all transactions. In more detail, MoC can be invoked in one of two modes. In *mint mode*, when given as input a desired initial supply $\mathbb{v}$, MoC deterministically derives a fresh unique identifier id for a new asset and stores $(\mathsf{id}, \mathbb{v}, v = \mathbb{v})$ in a *genesis record*. Later on, MoC can be invoked in *conserve mode*, where it inspects all records in a transaction having birth predicate equal to MoC and whose asset identifiers equal the identifier of the current record, and ensures that these records conserve asset values.

Users can program death predicates of such records to enforce conditions on how assets can be consumed, e.g., by realizing *conditional exchanges* with other counter-parties. Suppose that Alice wishes to exchange 100 units of an asset id_1 for 50 units of another asset 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_2. She then publishes out of band information about $\mathbf{r}$, and anyone can subsequently claim it by creating a transaction doing the exchange.

Since death predicates can be *arbitrary*, many different access policies can also be realized, e.g., to enforce that a transaction redeeming a record (a) must be authorized by two of three public keys, or (b) becomes valid only after a given amount of time, or (c) must reveal the pre-image of a hash.

One can generalize this basic example to show how the RNK can realize smart contract systems in which the transaction creator knows both the contract code being executed, as well as the (public and secret) state of the contract. At a high level, these contracts can be executed within a single transaction, or across multiple transactions, by storing suitable intermediate state/message data in record payloads, or by publishing that data in transaction memoranda (as plaintext or ciphertext as needed). We discuss in more detail below.

Example III.2 (smart contracts with caller-known state)**.** At

the highest level, smart contract systems operate over a set of individual contracts, each of which consists of a function (or collection of functions), some state variables, and some form of *address* that serves to uniquely identify the contract. The contract address ensures that the same code/functions can be deployed multiple times by different individuals, without two contracts inadvertently sharing state.[4] A standard feature of smart contract systems is that a contract can communicate with other contracts: that is, a contract can invoke a second smart contract as a subroutine, provided that the second contract provides an interface to allow this behavior. In our setting, we consider contracts in which the caller knows at least part of the state of each contract.

In this setting, one can use the records nano-kernel to realize basic smart contracts as follows. Each contract can be implemented as a function Φ_{sc}. The contract's state variables can be stored in one or more records such that each record $\mathbf{r}_i$ is labeled with Φ_{sc} as the birth and death predicate. Using this labeling, Φ_{sc} (via the RNK) can enforce that only it can update its state variables, thus fulfilling one requirement of a secure contract. Of course, while this serves to prevent *other functions* from updating the contract's state, it does not address the situation where multiple users wish to deploy different instances of the same function Φ_{sc}, each with isolated state. Fortunately (and validating our argument that the RNK realizes the *minimal* requirements needed for such a system), addressing this problem does not require changes to the RNK. Instead, one can devise the function Φ_{sc} so that it reasons over a unique contract address identifier id, which is recorded within the *payload* of every record.[5] The function Φ_{sc} can achieve contract state isolation by enforcing that each input and output state record considered by single execution of Φ_{sc} shares the same contract address.

To realize "inter-contract calls" between two functions Φ_{sc_1} and Φ_{sc_2}, one can use "ephemeral" records that communicate between the two functions. For example, if Φ_{sc_1} wishes to call Φ_{sc_2}, the caller may construct a record $\mathbf{r}_{\mathsf{e}}$ that contains the "arguments" to the called function Φ_{sc_2}, as well as the result of the function call. A transaction would then show that both Φ_{sc_1} and Φ_{sc_2} are satisfied.

The above example outlines how to implement a general smart contract system atop the RNK. We leave to future work the task of developing this outline into a full-fledged smart contract framework, and instead focus on constructing a scheme that implements the RNK, and on illustrating how to directly program the RNK to construct specific applications such as **private user-defined assets** and **private decentralized asset exchanges**. We discuss these applications in detail in Section V.

[4]In concrete implementations such as Ethereum [Woo17], contract identification is accomplished through unique contract addresses, each of which can be bound to a possibly non-unique `codeHash` that identifies the code of the program implementing the contract.

[5]This identifier can be generated in a manner similar to the asset identifier in Example III.1.

Remark III.3 (working in the UTXO model)**.** In the records nano-kernel, applications update their state by consuming records containing the old state, and producing new records that contain the updated state. This programming model is popularly known as the "unspent transaction output" (UTXO) model. This is in contrast to the "account-based" model which is used by many other smart contract systems [Goo14; Woo17; EOS18]. At present, it is not known how to efficiently achieve strong privacy properties in this model even for the simple case of privacy-preserving payments among any number of users, as we explain below.

In the account-based model, application state is stored in a persistent location associated with the application's account, and updates to this state are applied in-place. A smart contract that implements a currency in this model would store user balances in a persistent table T that maps user account identifiers to user balances. Transactions from a user A to another user B would then decrement A's balance in T and increment B's balance by a corresponding amount. A straightforward way to make this contract data-private (i.e., to hide the transaction value and the identities of A and B) would be to replace the user balances in T with *hiding commitments* to these balances; transactions would then update these commitments instead of directly updating the balances. However, while this hides transaction values, it does not hide user identities; to further hide these, every transaction would have to update *all* commitments in T, which entails a cost that grows linearly with the number of users. This approach is taken by zkLedger [NVV18], which enables private payments between a small number of known users (among other things).

Even worse, achieving function privacy when running multiple applications in such a system would require each transaction to hide which application's data was being updated, which means that the transaction would have to update the data of *all* applications at once, again severely harming the efficiency of the system.

In sum, it is unclear how to efficiently achieve strong data and function privacy in the account-based model when users can freely join and leave the system at any time. On the other hand, we show in this paper that these properties can be achieved in the UTXO model at a modest cost.

B. Decentralized private computation

A new cryptographic primitive. To realize a ledger-based system that supports privacy-preserving computations in the records nano-kernel, we introduce a new cryptographic primitive called *decentralized private computation* (DPC) schemes. Fig. 1 provides an overview of their interface; see the full version for a formal definition, including the ideal functionality that we use to express security.

Below we describe only a high-level sketch of our construction of a DPC scheme, and provide the details in Appendix B. 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

DPC.Setup
Input: security parameter 1^λ
Output: public parameters pp

DPC.GenAddress
Input: public parameters pp
Output: addr. key pair (apk, ask)

DPC.Execute$^{\mathbf{L}}$
Input:
- public parameters pp
- old { records $[\mathbf{r}_i]_1^m$; address secret keys $[\mathsf{ask}_i]_1^m$
- 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 $[\Phi_{\mathsf{d},j}]_1^n$
- auxiliary predicate input aux
- transaction memorandum memo

Output: new records $[\mathbf{r}_j]_1^n$ and transaction tx

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

Fig. 1: Algorithms of a DPC scheme.

of separate game-based definitions as in [BCG+14]). The full version contains our proof of security for this construction.

Data structures. A *record* is a data structure representing a unit of data in our system. Each record is associated with 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 itself consists of an address public key, a data payload, a birth predicate, a death predicate, and a serial number nonce and a *record commitment* that is a commitment to all of these attributes. The *serial number* of this record is the evaluation of a pseudorandom function, seeded with the record's address secret key and evaluated at the record's serial number nonce.

The record's commitment and serial number, which appear on the ledger when the record is created and consumed respectively, reveal *no* information about the record attributes. This follows from the hiding property of the commitment, and the pseudorandom nature 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. All the above is summarized in Fig. 2.

Records can be created and consumed via *transactions*, which represent state changes in the system. 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. 3.

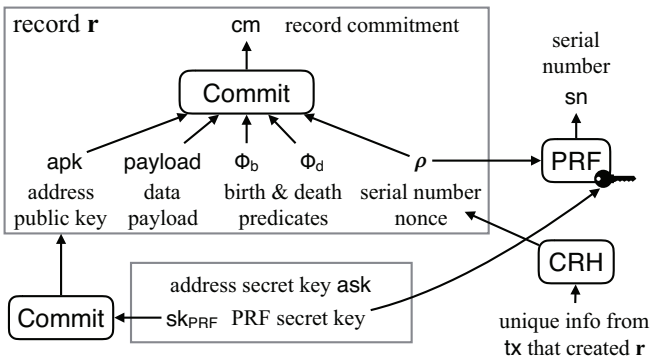

Fig. 2: Construction of a record.

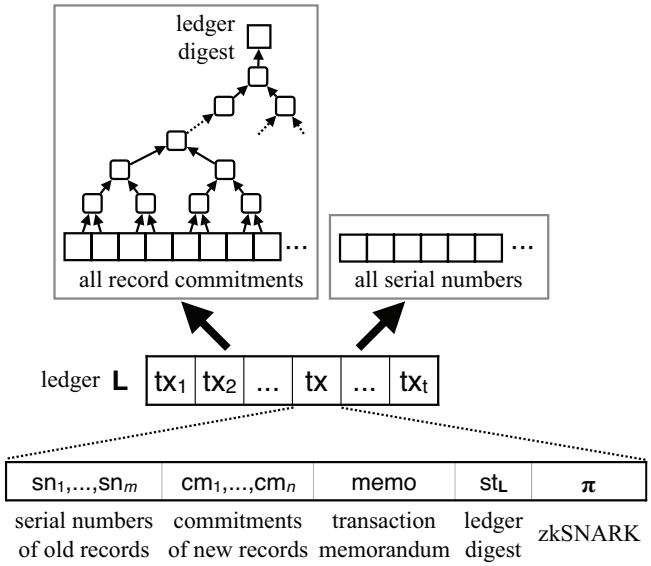

Fig. 3: Construction of a transaction.

System usage. To set up the system, a trusted party invokes DPC.Setup to produce the public parameters for the system. Later, users can invoke DPC.GenAddress to create address key pairs. In order to create and consume records, i.e., to produce a transaction, a user first 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. 4. 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 invoke DPC.Execute to produce a transaction containing a zero knowledge proof that attests to these conditions. See Fig. 5 for a summary of $\mathcal{R}_e$, the NP statement being proved.

Finally, nodes maintaining the ledger use DPC.Verify to check whether a candidate transaction is valid.

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).[6]

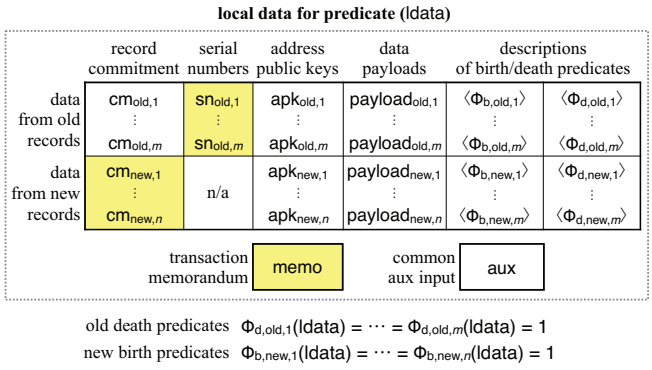

Fig. 4: Predicates receive local data.

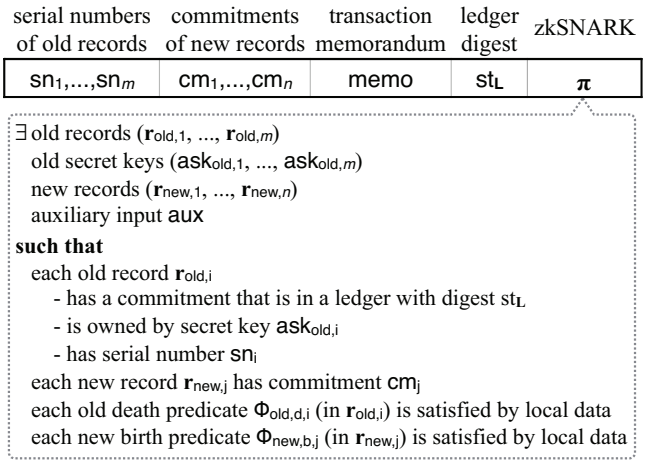

Fig. 5: The execute statement.

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 $\mathcal{R}_e$ being proved involves attesting the satisfiability of all (old) death and (new) birth predicates, and so we need to ensure that the time needed to verify the corresponding proof 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 IV), the foregoing modification suffices from a theoretical point of view.

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

IV. Achieving an efficient implementation

Our system Zexe (*Zero knowledge EXEcution*) implements our construction of a DPC scheme (see Section III-B and Appendix B). Achieving efficiency in our system required overcoming several challenges. Below we adopt a "problem-solution" format to highlight some of these challenges and explain how we addressed them.

Problem 1: universality is expensive. The NP statement $\mathcal{R}_e$ 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; BCT+14; WSR+15; BCT+17]. Using such "heavy duty" proof systems would make the system costly for all users, including those that produce transactions that attest to simple inexpensive predicates.

Solution 1: recursive proof verification. We address this problem by relying on one layer of *recursive proof composition* [Val08; BCC+13]. Instead of tasking the NP statement with directly checking user-defined predicates, we only task it with checking *succinct proofs* attesting to the satisfaction of the same. Checking these succinct predicate proofs is a (relatively) inexpensive computation that is fixed for *all* predicates, and which can be "hardcoded" in $\mathcal{R}_e$. Since the single succinct proof produced for $\mathcal{R}_e$ does not reveal information about the predicate proofs (thanks to zero knowledge), the predicate proofs do *not* have to hide what predicate was checked, and hence can be specialized for particular user-defined predicates. This approach further ensures that a user only has to incur the cost of proving satisfiability of the specific predicates involved in her transactions, regardless of the complexity of predicates used by other users in their transactions.

Problem 2: recursion is expensive. Recursive proof composition has so far been empirically demonstrated for pairing-based SNARKs [BCT+17] as these have proofs that 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 statements 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 $\mathcal{R}_e$ via arithmetic circuits over $\mathbb{F}_r$. In turn, since the SNARK verifier is part of $\mathcal{R}_e$, 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 is expensive, and one cannot avoid simulation by picking E such that $q = r$ [BCT+17].

Prior work overcomes this by using *multiple* curves [BCT+17]. Specifically, Ben-Sasson et al. distribute the recursion across a two-cycle of pairing-friendly elliptic curves, which is a pair of prime-order curves E_1 and E_2 such that the size of one's base field is the order of the other's subgroup. This ensures that a SNARK over E_1 can be verified by a

SNARK over E_2, and vice versa. However, known cycles are inefficient at 128 bits of security [BCT+17; CCW19].

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), but rather only a "two-chain" of two curves E_1 and E_2 such that the size of the base field of E_1 is the size of the prime order subgroup of E_2. We can use the Cocks–Pinch method [FST10] to set up such a bounded recursion [BCT+17]. We now elaborate on this.

First, we pick a pairing-friendly elliptic curve E_1 that not only is suitable for 128 bits of security, but moreover, enables efficient SNARK provers at *both* levels of the recursion. Namely, letting p be the prime order of E_1's 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 E_1, while the condition on $\mathbb{F}_p$ ensures efficient proving for SNARKs that verify proofs over E_1. In light of the above, we set E_1 to be E_{BLS}, a curve from the Barreto–Lynn–Scott (BLS) family [BLS02; CLN11] with embedding degree 12. This family can be implemented at 128 bits of security very efficiently [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.[7]

Next we use the Cocks–Pinch method to pick a pairing-friendly elliptic curve $E_2 = E_{\mathsf{CP}}$ 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 [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 predicate proofs are valid (along with the remaining NP statement's checks). Because the two curves have "matching" fields, proofs over E_{BLS} are efficiently verifiable.

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 is $2\times$ larger than 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 performed on 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 $\mathcal{R}_{\mathsf{CP}}$, with the latter containing just the proof check and the former containing all other checks (see the full version for details on these). 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.

Problem 4: the NP relations have many checks. Even using E_{CP} only for SNARK verification and E_{BLS} for all other checks does not suffice: the NP relations $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$ still have to perform expensive checks like verifying Merkle tree authentication paths and commitment openings, and evaluating pseudorandom functions and collision resistant functions. Similar NP relations, like the one in Zerocash [BCG+14], require upwards of *four million gates* to express such checks, resulting in high latencies for producing transactions and large public parameters for the system.

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 [HBH+18]). 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.) Overall, we obtain highly optimized realizations of all checks in Fig. 5.

A note on deploying Zexe with trusted setup. DPC schemes include a setup algorithm that specifies how to sample public parameters for the scheme. The setup algorithm in our DPC construction (see Section III-B) simply consists of running the setup algorithms for the various cryptographic building blocks that we rely on (like NIZKs). However, this can be a challenge for deployment because the entity performing the setup may be able to break certain security properties of the scheme by acting maliciously.

While one can mitigate this by using primitives that have a *transparent setup* (one that uses only public randomness), the efficiency considerations mentioned above drive our implemented system to use pairing-based zkSNARKs whose setup is *not* transparent (all other primitives we use are transparent). We thus discuss below how to perform this setup when deploying our implemented system.

Recall that prior zkSNARK deployments have used secure multiparty computation [BCG+15; Zcab; BGM17; BGG18], so that the sampled public parameters are secure as long as

[7] We achieve this 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].

even a single participating party is honest. One can use this technique to sample "master" parameters for SNARKs for the NP relations $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$. Since these public parameters do *not* depend on any user-defined functions, they can be sampled once and for all regardless of which applications will run over the system. Note that these public parameters must be trusted by *everyone*, because if they were compromised then the security (but not privacy) of *all* applications running over the system would be compromised as well.

In addition to these "master" parameters, application developers must also sample "application" parameters. These are the parameters corresponding to the predicates comprising an application. Unlike "master" parameters, "application" parameters can be sampled as applications are developed and deployed. Furthermore, users only need to trust the parameters needed by applications that the user cares about; compromised parameters for other applications will not affect (the security and privacy of) the user's applications.

Very recent works [MBK+19; CFQ19; CHM+19; GWC19] have proposed pairing-based SNARKs that have a universal setup that can be used for *any* circuit. Once such SNARK constructions mature into efficient implementations, our system can be easily modified to use these instead of [GM17] to mitigate the above concerns, as both our construction and implementation make use of the underlying SNARKs in a modular manner.

V. Applications

We describe example applications of DPC schemes by showing how to "program" these within the records nano-kernel. We focus on financial applications of smart contract systems as these are not only popular, but also demand strong privacy. We begin in Section V-A by describing how to enable users to privately create and transact with **custom user-defined assets** (expanding on Example III.1). We then describe in Section V-B how to realize **private DEXs**, which enable users to privately trade these assets while retaining custody of the same. These descriptions are a high-level sketch; further details are available in the full version.

A. 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). We show below how to express such custom assets via the records nano-kernel.

We consider records whose payloads encode: an asset identifier id, the initial asset supply $\mathtt{v}$, a value v, and application-dependent data c (we will use this in Section V-B). We fix the birth predicate in all such records to be a *mint-or-conserve* function MoC that is responsible for asset minting and conservation. In more detail, the birth predicate MoC can be invoked in two modes, *mint mode* or *conserve mode*.

When invoked in mint mode, MoC creates the initial supply $\mathtt{v}$ of the asset in a single output record by deterministically deriving a fresh, globally-unique identifier id for the asset, and storing the tuple $(\mathsf{id}, \mathtt{v}, \mathtt{v}, \bot)$ in the record's payload. The predicate MoC also ensures that in the given transaction contains no other non-dummy input or output records. If MoC is invoked in mint mode in a different transaction, a *different* identifier id is created, ensuring that multiple assets can be distinguished even though anyone can use MoC as the birth predicate of a record.

When invoked in conserve mode, MoC inspects all records in a transaction whose birth predicates all equal MoC (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 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.

The full version contains pseudocode for MoC.

B. Decentralized exchanges

We describe how to use death predicates that enforce custom-access policies to build *privacy-preserving decentralized exchanges*, which allow users to exchange custom assets with strong privacy guarantees while retaining full custody of these assets. We proceed by first providing background on centralized and decentralized exchanges. Then, we formulate desirable privacy properties for decentralized exchanges. Finally, we describe constructions that achieve these properties.

Motivation. Exchanging digital assets is a compelling use case of ledger-based systems. A straightforward method to exchange such assets is via a *centralized exchange*: users entrust the exchange with custody of their assets via an on-chain transaction so that subsequent trades require only off-chain modifications in the exchange's internal database. To "exit", users can request an on-chain transaction that transfers their assets from the exchange to the user. Examples of such exchanges include Coinbase [Coi] and Binance [Bin]. This centralized architecture is *efficient*, because trades are recorded only in the exchange's off-chain database, and relatively *private*, because only the exchange knows the details of individual trades. However, it also has a serious drawback: having given up custody of their assets, users are exposed to the risk of security breaches and fraud by the exchange [PA14; De18; Zha18; Cim18].

In light of this, *decentralized exchanges* (DEXs) have been proposed as an alternative means of exchanging assets that enable users to retain custody of their assets. However, existing DEX constructions have poor efficiency and privacy guarantees. Below we describe how we can provide strong privacy for DEXs (and leave improving the efficiency of DEXs to future work).

DEX architectures. There are different DEX architectures with different trade-offs; see [Pro18] for a survey. In the following, we consider DEX architectures where the exchange has no state or maintains its state off-chain.[8] Here we focus on

[8]This is in contrast to DEX architectures that involve, say, a smart contract that stores on-chain the standing orders of all users.

one such category of DEXs, namely *intent-based DEXs*; we discuss other kinds of DEXs in the full version.

In intent-based DEXs, the DEX maintains an index, which is a table where makers publish their intention to trade (say, a particular asset pair) *without committing any assets*. A taker interested in a maker's intention to trade can directly communicate with the maker to agree on terms. They can jointly produce a transaction for the trade, to be broadcast for on-chain processing. An example of such a DEX is AirSwap [Air]. An attractive feature of intent-based DEXs is that they reduce exposure to front-running because the information required for front-running (like prices or identities of the involved parties) has been finalized by the time the transaction representing the trade is broadcast for processing.

Privacy shortcomings and goals. While the foregoing DEX architecture offers attractive security and functionality, it does *not* provide strong privacy guarantees. First, each transaction reveals information about the corresponding trade, such as the assets and amounts that were exchanged. Prior work [BDJ+17; BBD+17; EMC19; DGK+19] shows that such leakage enables front-running that harms user experience and market transparency, and proposes mitigations that, while potentially useful, do not provide strong privacy guarantees. Even if one manages to hide these trade details, transactions in existing DEXs *also* reveal the identities of transacting parties. Onlookers can use this information to extract trading patterns and frequencies of users. This reduces the privacy of users, violates the fungibility of assets, and increases exposure to front-running, because onlookers can use these patterns to infer when particular assets are being traded.

These shortcomings motivate the following privacy goals for DEXs. Throughout, we assume that an order is defined by the pair of assets to be exchanged, and their exchange rates.

1) *Trade confidentiality:* No efficient adversary $\mathcal{A}$ should be able to learn the trade details (i.e., the asset pairs or amounts involved) of completed or cancelled trades.
2) *Trade anonymity:* No efficient adversary $\mathcal{A}$ should be able to learn the identities of the maker and taker.

A protocol that achieves trade confidentiality and trade anonymity against an adversary $\mathcal{A}$ is secure against front-running by $\mathcal{A}$. We now describe how to construct an intent-based DEX that achieves trade confidentiality and anonymity.[9]

Record format. Recall from Section V-A that records representing units of an asset have payloads of the form $(\mathsf{id}, \mathbb{v}, v, c)$, where id is the asset identifier, $\mathbb{v}$ is the initial asset supply, v is the asset amount, and c is arbitrary auxiliary information. In the following, we use records that, in addition to the mint-or-conserve birth predicate MoC, have an *exchange-or-cancel* death predicate EoC. Informally, EoC allows a record $\mathbf{r}$ to be consumed either by exchanging it for $v^\star$ units of an asset with birth predicate $\Phi^\star_{\mathsf{b}}$ and identifier $\mathsf{id}^\star$ ($\mathsf{id}^\star$, $\Phi^\star_{\mathsf{b}}$ and $v^\star$ are specified in c), or by "cancelling" the exchange and instead sending new records with $\mathbf{r}$'s asset identifier to an address $\mathsf{apk}^\star$ (also specified in c). The information required for the exchange includes the asset's birth predicate in addition to its identifier, as it enables users to interact with assets that have birth predicate different from MoC. See the full version for detailed pseudocode for EoC.

Private intent-based DEXs. We describe an intent-based DEX that hides all information about an order and the involved parties:

1) A maker M can publish to the index an intention to trade, which is a tuple $(\mathsf{id}_A, \mathsf{id}_B, \mathsf{pk}_\mathsf{M})$ to be interpreted as: "I want to buy assets with identifier id_B in exchange for assets with identifier id_A. Please contact me using the encryption public key pk_M if you would like to discuss the terms."
2) A taker T who is interested in this offer can use pk_M to privately communicate with M and agree on the terms of the trade (the form of communication is irrelevant). Suppose that T and M agree that T will give 10 units of asset id_B to M and will receive 5 units of asset id_A from M.
3) The taker T creates a new record $\mathbf{r}$ with payload $(\mathsf{id}_B, \mathbb{v}_B, 10, c)$ for auxiliary data $c = (\mathsf{id}_A, 5, \mathsf{apk}_{\mathsf{new}})$, and with death predicate EoC. Then T sends $\mathbf{r}$ (along with the information necessary to redeem $\mathbf{r}$) to M.
4) If M has a record worth 5 units of asset id_A, she can use T's message to construct a DPC transaction that consumes $\mathbf{r}$ and produces appropriate new records for M and T, thereby completing the exchange.

The record $\mathbf{r}$ produced by the taker T can be redeemed by M only via an appropriate record in exchange. If M does not possess such a record, T can cancel the trade (at any time) and retrieve his funds by satisfying the "cancel" branch of the predicate EoC (which requires knowing the secret key corresponding to $\mathsf{apk}_{\mathsf{new}}$).

Note that regardless of whether the trade was successful or not, this protocol achieves trade anonymity and trade confidentiality against all parties (including the index operator). Indeed, the only information revealed in the final transaction is that some records were consumed and others created; no information is revealed about M, T, the assets involved in the trade (id_A and id_B), or the amounts exchanged.

VI. System implementation

We now summarize our implementation of DPC schemes in our system named named **Zexe** (*Zero knowledge EXEcution*).[10] Zexe follows the strategy described in Section IV, and 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 DPC construction. Our codebase, like our construction, is written in terms of abstract building blocks, which allows to easily switch between different

[9]Throughout, we assume that users interact with index operators via anonymous channels. (If this is not the case, operators can use network information to link users across different interactions regardless of any cryptographic solutions used.).

[10]The code is available at https://github.com/scipr-lab/zexe.

instantiations of the building blocks. In the rest of this section we describe the efficient instantiations used in the experiments reported in Section VII.

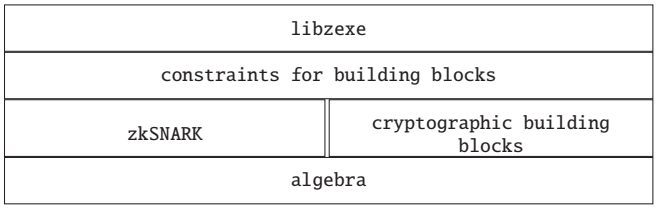

Fig. 6: Stack of libraries comprising Zexe.

Pseudorandom function. Fixing key length and input length at 256 bits, we instantiate PRF using the Blake2s hash function [ANW+13]: $\mathsf{PRF}_k(x) := \mathsf{b2s}(k \| x)$ for $k, x \in \{0,1\}^{256}$.

Elliptic curves. Our implementation strategy (see Section IV) 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 Fig. 7; 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 IV 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. Concretely, we rely on the simulation-extractable zkSNARK of Groth and Maller [GM17], used over the pairing-friendly elliptic curves E_{BLS} (for proving $\mathcal{R}_{\mathsf{BLS}}$ and predicates' satisfiability) and E_{CP} (for proving $\mathcal{R}_{\mathsf{CP}}$).

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 SampleGrp, which on input a security parameter λ (represented in unary), outputs a tuple $(\mathbb{G}, q, g)$ that 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.

Commitments. We instantiate (plain and) trapdoor commitments via Pedersen commitments over $\mathbb{G}$. These 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 $\mathbb{G}$. Collision resistance follows from hardness of the discrete-logarithm problem [MRK03].

VII. System evaluation

In Section VII-A we evaluate individual cryptographic building blocks. In Section VII-B we evaluate the cost of NP relations expressed as constraints, as required by the underlying zkSNARK. In Section VII-C we evaluate the running time of DPC algorithms. In Section VII-D 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 GHz with 252 GB of RAM.

A. 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 µs per constraint (with 12 threads), while the GM17 verifier takes $250n\,\mu\mathrm{s} + 9.5\,\mathrm{ms}$ 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 µs per constraint and $1.6n\,\mathrm{ms} + 34\,\mathrm{ms}$.

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.

B. 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 large prime field. Our 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. 10. More precisely, for efficiency reasons explained in Section IV, we split $\mathcal{R}_{\mathsf{e}}$ into the two NP relations $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$, which we prove via zkSNARKs over the pairing-friendly curves E_{BLS} and E_{CP}, respectively.

Table III 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 IV 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. We also report the cumulative number of constraints when setting $m := 2$ and $n := 2$ because this is a representative instantiation of m and n that enables useful applications.

C. DPC algorithms

In Table I we report the running times of algorithms in our DPC implementation for two input and two output records (i.e., $m := 2$ and $n := 2$). Note that for Execute and Verify, we have excluded costs of ledger operations (such as retrieving an authentication path or scanning for duplicate serial numbers)

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

Fig. 7: The elliptic curves $E_{\mathsf{BLS}}, E_{\mathsf{CP}}, E_{\mathsf{Ed/BLS}}, E_{\mathsf{Ed/CP}}$. See Appendix A for details of the underlying fields.

because these depend on how a ledger is realized, which is orthogonal to our work. Also, we assume that Execute receives as inputs the application-specific 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 describe the cost of doing so for representative applications in Section VII-E.

Observe 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 on 1 thread. Overall, we learn that Execute takes less 1 min, Verify takes roughly 50 ms, and both Setup and Execute use less than 5 GB of RAM. These costs are comparable to those of similar systems such as Zerocash [BCG+14] and Hawk [KMS+16].

D. 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. 7); the corresponding secret key is 64 bytes and consists of a PRF seed (32 bytes) and commitment randomness (32 bytes).

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, a transaction with m input and n output records is $32m + 32n + 840$ bytes.

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).

E. Applications

We do not report total costs for producing transactions for the applications in Section V because the additional application-specific costs are *negligible* compared to the base cost reported in Table I. This is because all application-specific proofs are produced over the efficient elliptic curve E_{BLS}, and moreover, for each application we consider, the heaviest computation checked by these proofs is the relatively lightweight one of opening the local data commitment; the remaining costs consist of a few cheap range and equality checks. Indeed, with two input and two output records, these applications require fewer than $35,000$ constraints (compared to over $350,000$ for $\mathcal{R}_{\mathsf{BLS}}$ and $\mathcal{R}_{\mathsf{CP}}$), and producing the corresponding proofs takes tens of milliseconds (compared to tens of seconds for the base cost of DPC.Execute).

Setup	109.62 s
GenAddress	380 µs
Execute	52.5 s
Verify	46 ms

TABLE I: Cost of DPC algorithms for 2 inputs and 2 outputs.

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

TABLE II: Size of a DPC transaction (in bytes).

Breakdown of the number of constraints with m input and n output records:		
Per input record	Total	117699
	Enforce validity of:	
	Merkle tree path	81824
	Address key pair	3822
	Serial number computation	22301
	Record commitment	9752
Per output record	Total	15427
	Enforce validity of:	
	Serial number nonce	5417
	Record commitment	10010
Other	Enforce validity of:	
	Predicate commitment	$21792 \cdot \lceil \frac{3}{4}(m+n) + \frac{1}{2} \rceil$
	Local data commitment	$7168 \cdot m + 6144 \cdot n$
	Miscellaneous	7368
Total with 2 inputs and 2 outputs ($m = n = 2$)		**387412**

TABLE III: Number of constraints for $\mathcal{R}_{\mathsf{BLS}}$.

Breakdown of the number of constraints with m input and n output records:		
Per input record	Total	87569
	Enforce validity of:	
	Death predicate ver. key	45827
	Death predicate proof	41742
Per output record	Total	87569
	Enforce validity of:	
	Birth predicate ver. key	45827
	Birth predicate proof	41742
Other	Enforce validity of:	
	Predicate commitment	$21792 \cdot \lceil \frac{3}{4}(m+n) + \frac{1}{2} \rceil$
	Miscellaneous	1780
Total with 2 inputs and 2 outputs ($m = n = 2$)		**439224**

TABLE IV: Number of constraints for $\mathcal{R}_{\mathsf{CP}}$.

Appendix A
Details of elliptic curves used in Zexe

In Fig. 8 we report details of the base fields and prime orders of the elliptic curves $E_{\mathsf{Ed/BLS}}$, E_{BLS}, $E_{\mathsf{Ed/CP}}$ and E_{CP}.

Appendix B
Construction of decentralized private computation schemes

We describe our construction of a DPC scheme. In Section B-A we introduce the building blocks that we use, and in Section B-B we describe each algorithm in the scheme. The security proof is provided in the full version.

A. 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 to anyone who does not know the x.

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

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

We also use a *trapdoor* commitment scheme $\mathsf{TCM} = (\mathsf{Setup}, \mathsf{Commit})$, with the same syntax as above. Auxiliary algorithms (beyond those in CM) 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.

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 $\mathsf{pp}_{\mathsf{NIZK}}$.
- *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 B.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 VI.

B. Algorithms

Pseudocode for our construction of a DPC scheme is in Fig. 9. The construction involves invoking zero knowledge proofs for the NP relation $\mathcal{R}_{\mathsf{e}}$ described in Fig. 10. 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 $\mathsf{CM.Setup}$, $\mathsf{TCM.Setup}$, $\mathsf{CRH.Setup}$, and $\mathsf{NIZK.Setup}$ to obtain (plain and trapdoor) commitment public parameters $\mathsf{pp}_{\mathsf{CM}}$ and $\mathsf{pp}_{\mathsf{TCM}}$, CRH public parameters $\mathsf{pp}_{\mathsf{CRH}}$, and NIZK public parameters for the NP relation $\mathcal{R}_{\mathsf{e}}$ (see Fig. 10). It then outputs $\mathsf{pp} := (\mathsf{pp}_{\mathsf{CM}}, \mathsf{pp}_{\mathsf{TCM}}, \mathsf{pp}_{\mathsf{CRH}}, \mathsf{pp}_{\mathsf{e}})$.

Address creation. $\mathsf{DPC.GenAddress}$ constructs an address key pair as follows. The address secret key $\mathsf{ask} = (\mathsf{sk}_{\mathsf{PRF}}, r_{\mathsf{pk}})$ consists of a seed $\mathsf{sk}_{\mathsf{PRF}}$ for the pseudorandom function PRF and commitment randomness r_{pk}. The address public key apk is a hiding commitment to $\mathsf{sk}_{\mathsf{PRF}}$ with randomness r_{pk}.

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.

$\mathsf{DPC.ConstructRecord}(\mathsf{pp}, \mathsf{apk}, \mathsf{payload}, \Phi_{\mathsf{b}}, \Phi_{\mathsf{d}}, \rho) \to (\mathbf{r}, \mathsf{cm})$
1) Sample new commitment **randomness** r.
2) Assemble new record **commitment contents**: $m := (\mathsf{apk} \| \mathsf{payload} \| \Phi_{\mathsf{b}} \| \Phi_{\mathsf{d}} \| \rho)$.
3) Construct new record **commitment**: $\mathsf{cm} \leftarrow \mathsf{TCM.Commit}(\mathsf{pp}_{\mathsf{TCM}}, m; r)$.
4) Assemble new **record**
$$\mathbf{r} := \begin{pmatrix} \text{address public key } \mathsf{apk} & \text{payload } \mathsf{payload} & \text{comm. rand. } r \\ \text{serial number nonce } \rho & \text{predicates } (\Phi_{\mathsf{b}}, \Phi_{\mathsf{d}}) & \text{commitment } \mathsf{cm} \end{pmatrix}.$$
5) 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. 10).

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 $\mathbf{r}_i \in [\mathbf{r}_i]_1^m$:

prime	value	size in bits	2-adicity
s	0x4aad957a68b2955982d1347970dec005293a3afc43c8afeb95aee9ac33fd9ff	251	1
r	0x12ab655e9a2ca55660b44d1e5c37b00159aa76fed00000010a11800000000001	253	47
t	0x35c748c2f8a21d58c760b80d94292763445b3e601ea271e1 d75fe7d6eeb84234066d10f5d893814103486497d95295	374	2
p	0x1ae3a4617c510eac63b05c06ca1493b1a22d9f300f5138f1 ef3622fba094800170b5d44300000008508c00000000001	377	46
q	0x3848c4d2263babf8941fe959283d8f526663bc5d176b746a f0266a7223ee72023d07830c728d80f9d78bab3596c8617c57 9252a3fb77c79c13201ad533049cfe6a399c2f764a12c4024b ee135c065f4d26b7545d85c16dfd424adace79b57b942ae9	782	3

Fig. 8: The elliptic curves $E_{\mathsf{BLS}}, E_{\mathsf{CP}}, E_{\mathsf{Ed/BLS}}, E_{\mathsf{Ed/CP}}$.

  - (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 $\mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, j \| \mathsf{sn}_1 \| \ldots \| \mathsf{sn}_m)$;
  - $\mathbf{r}_j$*'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 $\mathsf{ask} = (\mathsf{sk}_{\mathsf{PRF}}, r_{\mathsf{pk}})$ is derived by evaluating PRF at $\mathbf{r}$'s serial number nonce ρ with seed $\mathsf{sk}_{\mathsf{PRF}}$. 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.

References

[ADM+14a] M. Andrychowicz et al. "Fair Two-Party Computations via Bitcoin Deposits". In: FC '14.

[ADM+14b] M. Andrychowicz et al. "Secure Multiparty Computations on Bitcoin". In: SP '14.

[AFK+12] D. F. Aranha et al. "Implementing Pairings at the 192-Bit Security Level". In: Pairing '12.

[Air] "AirSwap". https://www.airswap.io/. Accessed 2018-12-27.

[AKR+13] E. Androulaki et al. "Evaluating User Privacy in Bitcoin". In: FC '13.

[ANW+13] J. Aumasson et al. "BLAKE2: Simpler, Smaller, Fast as MD5". In: ACNS '13.

[BAZ+19] B. Bünz et al. "Zether: Towards Privacy in a Smart Contract World". https://crypto.stanford.edu/~buenz/papers/zether.pdf.

[BBB+18] B. Bünz et al. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: S&P '18.

[BBD+17] I. Bentov et al. "The Cost of Decentralization in 0x and EtherDelta". http://hackingdistributed.com/2017/08/13/cost-of-decent/. Accessed 2019-01-03.

[BCC+13] N. Bitansky et al. "Recursive Composition and Bootstrapping for SNARKs and Proof-Carrying Data". In: STOC '13.

[BCC+16] J. Bootle et al. "Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting". In: EUROCRYPT '16.

[BCG+13] E. Ben-Sasson et al. "SNARKs for C: verifying program executions succinctly and in zero knowledge". In: CRYPTO '13.

[BCG+14] E. Ben-Sasson et al. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: SP '14.

[BCG+15] E. Ben-Sasson et al. "Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs". In: SP '15.

[BCS16] E. Ben-Sasson et al. "Interactive Oracle Proofs". In: TCC '16-B.

[BCT+14] E. Ben-Sasson et al. "Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture". In: USENIX '14.

[BCT+17] E. Ben-Sasson et al. "Scalable Zero Knowledge Via Cycles of Elliptic Curves". In: *Algorithmica* (2017).

[BDJ+17] L. Breidenbach et al. "To Sink Frontrunners, Send in the Submarines". http://hackingdistributed.com/2017/08/28/submarine-sends/. Accessed 2019-01-03.

[BGG18] S. Bowe et al. "A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK".

[BGM17] S. Bowe et al. "Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model". ePrint Report 2017/1050.

[Bin] "Binance". https://www.binance.com/. Accessed 2019-01-03.

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

[BKM17] I. Bentov et al. "Instantaneous Decentralized Poker". In: ASIACRYPT '17.

[BLS02] P. Barreto et al. "Constructing Elliptic Curves with Prescribed Embedding Degrees". In: SCN '02.

DPC.Setup
Input: security parameter 1^λ
Output: public parameters pp

1) Generate **commitment parameters**: $\mathsf{pp}_{\mathsf{CM}} \leftarrow \mathsf{CM.Setup}(1^\lambda)$, $\mathsf{pp}_{\mathsf{TCM}} \leftarrow \mathsf{TCM.Setup}(1^\lambda)$.
2) Generate **CRH parameters**: $\mathsf{pp}_{\mathsf{CRH}} \leftarrow \mathsf{CRH.Setup}(1^\lambda)$.
3) Generate **NIZK parameters for** $\mathcal{R}_{\mathsf{e}}$ (see Figure 10): $\mathsf{pp}_{\mathsf{e}} \leftarrow \mathsf{NIZK.Setup}(1^\lambda, \mathcal{R}_{\mathsf{e}})$.
4) Output $\mathsf{pp} := (\mathsf{pp}_{\mathsf{CM}}, \mathsf{pp}_{\mathsf{TCM}}, \mathsf{pp}_{\mathsf{CRH}}, \mathsf{pp}_{\mathsf{e}})$.

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

1) Sample secret key $\mathsf{sk}_{\mathsf{PRF}}$ for pseudorandom function PRF.
2) Sample randomness r_{pk} for commitment scheme CM.
3) Set **address public key** $\mathsf{apk} := \mathsf{CM.Commit}(\mathsf{pp}_{\mathsf{CM}}, \mathsf{sk}_{\mathsf{PRF}};\ r_{\mathsf{pk}})$.
4) Set **address secret key** $\mathsf{ask} := (\mathsf{sk}_{\mathsf{PRF}}, r_{\mathsf{pk}})$.
5) Output $(\mathsf{apk}, \mathsf{ask})$.

DPC.Execute$^{\mathbf{L}}$
Input:
- public parameters 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 aux
- transaction memorandum memo

Output: new records $[\mathbf{r}_j]_1^n$ and transaction 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 $\begin{pmatrix} \text{address public key} & \mathsf{apk}_i & \text{payload} & \mathsf{payload}_i & \text{comm. rand.} & r_i \\ \text{serial number nonce} & \rho_i & \text{predicates} & (\Phi_{\mathsf{b},i}, \Phi_{\mathsf{d},i}) & \text{commitment} & \mathsf{cm}_i \end{pmatrix}$.
   b) If $\mathsf{payload}_i.\mathsf{isDummy} = 1$, set **ledger membership witness** $\mathrm{w}_{\mathbf{L},i} := \bot$.
   If $\mathsf{payload}_i.\mathsf{isDummy} = 0$, compute **ledger membership witness** for commitment: $\mathrm{w}_{\mathbf{L},i} \leftarrow \mathbf{L}.\mathsf{Prove}(\mathsf{cm}_i)$.
   c) Parse address secret key ask_i as $(\mathsf{sk}_{\mathsf{PRF},i}, r_{\mathsf{pk},i})$.
   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**: $\rho_j := \mathsf{CRH.Eval}(\mathsf{pp}_{\mathsf{CRH}}, j \| \mathsf{sn}_1 \| \ldots \| \mathsf{sn}_m)$.
   b) Construct **new record**: $(\mathbf{r}_j, \mathsf{cm}_j) \leftarrow \mathsf{DPC.ConstructRecord}(\mathsf{pp}_{\mathsf{TCM}}, \mathsf{apk}_j, \mathsf{payload}_j, \Phi_{\mathsf{b},j}, \Phi_{\mathsf{d},j}, \rho_j)$.
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** $\mathcal{R}_{\mathsf{e}}$: $\mathbb{w}_{\mathsf{e}} := ([\mathbf{r}_i]_1^m, [\mathrm{w}_{\mathbf{L},i}]_1^m, [\mathsf{ask}_i]_1^m, [\mathbf{r}_j]_1^n, \mathsf{aux})$.
6) Generate **proof for** $\mathcal{R}_{\mathsf{e}}$: $\pi_{\mathsf{e}} \leftarrow \mathsf{NIZK.Prove}(\mathsf{pp}_{\mathsf{e}}, \mathbb{x}_{\mathsf{e}}, \mathbb{w}_{\mathsf{e}})$.
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 $([\mathbf{r}_j]_1^n, \mathsf{tx})$.

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

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) 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** $\mathcal{R}_{\mathsf{e}}$: $\mathsf{NIZK.Verify}(\mathsf{pp}_{\mathsf{e}}, \mathbb{x}_{\mathsf{e}}, \pi_{\mathsf{e}}) = 1$.

Fig. 9: Construction of a DPC scheme.

[Bow17a] S. Bowe. "Bellman". URL: https://github.com/zkcrypto/bellman.

[Bow17b] S. Bowe. "Pairing". URL: https://github.com/zkcrypto/pairing.

[CCW19] A. Chiesa et al. "On Cycles of Pairing-Friendly Elliptic Curves". In: *SIAM Journal on Applied Algebra and Geometry* (2019).

[CFQ19] M. Campanelli et al. "LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs". ePrint Report 2019/142.

[Cha14] Chainalysis. "Chainalysis Inc". https://chainalysis.com/.

[CHM+19] A. Chiesa et al. "Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS". ePrint Report 2019/1047.

[Cim18] C. Cimpanu. "Zaif cryptocurrency exchange loses $60 million in recent hack". https://www.zdnet.com/article/zaif-cryptocurrency-exchange-loses-60-million-in-july-hack/. Accessed 2018-12-27.

[CLN11] C. Costello et al. "Attractive Subfamilies of BLS Curves for Implementing High-Security Pairings". In: INDOCRYPT '11.

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

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

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: $\mathsf{apk}_i = \mathsf{CM}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{CM}}, \mathsf{sk}_{\mathsf{PRF},i};\ r_{\mathsf{pk},i})$ and $\mathsf{ask}_i = (\mathsf{sk}_{\mathsf{PRF},i}, r_{\mathsf{pk},i})$.
   - 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: $\mathsf{cm}_i = \mathsf{TCM}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{TCM}}, \mathsf{apk}_i \| \mathsf{payload}_i \| \Phi_{\mathsf{b},i} \| \Phi_{\mathsf{d},i} \| \rho_i;\ r_i)$.
   - The death predicate $\Phi_{\mathsf{d},i}$ is satisfied by 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: $\rho_j = \mathsf{CRH}.\mathsf{Eval}(\mathsf{pp}_{\mathsf{CRH}}, j \| \mathsf{sn}_1 \| \ldots \| \mathsf{sn}_m)$.
   - The new record commitment cm_j is valid: $\mathsf{cm}_j = \mathsf{TCM}.\mathsf{Commit}(\mathsf{pp}_{\mathsf{TCM}}, \mathsf{apk}_j \| \mathsf{payload}_j \| \Phi_{\mathsf{b},j} \| \Phi_{\mathsf{d},j} \| \rho_j;\ r_j)$.
   - The birth predicate $\Phi_{\mathsf{b},j}$ is satisfied by local data: $\Phi_{\mathsf{b},j}(j \| \mathsf{ldata}) = 1$.

Fig. 10: The execute NP relation $\mathcal{R}_{\mathsf{e}}$.

[Coi] "Coinbase". https://www.coinbase.com/. Accessed 2019-01-03.

[Cos12] C. Costello. "Particularly Friendly Members of Family Trees". ePrint Report 2012/072.

[CRR11] R. Canetti et al. "Practical delegation of computation using multiple servers". In: CCS '11.

[CRR13] R. Canetti et al. "Refereed delegation of computation". In: *Information and Computation* (2013).

[CZJ+17] E. Cecchetti et al. "Solidus: Confidential Distributed Ledger Transactions via PVORM". In: CCS '17.

[CZK+18] R. Cheng et al. "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution". arXiv cs.CR/1804.05141.

[DDO+01] A. De Santis et al. "Robust Non-interactive Zero Knowledge". In: CRYPTO '01.

[De18] N. De. "Coincheck Confirms Crypto Hack Loss Larger than Mt. Gox". https://www.coindesk.com/coincheck-confirms-crypto-hack-loss-larger-than-mt-gox. Accessed 2018-12-27.

[DGK+19] P. Daian et al. "Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges". arXiv cs.CR/1904.05234.

[DSC+15] T. T. A. Dinh et al. "M2R: Enabling Stronger Privacy in MapReduce Computation". In: USENIX Security '15.

[Ell13] Elliptic. "Elliptic Enterprises Limited". https://www.elliptic.co/.

[EMC19] S. Eskandari et al. "SoK: Transparent Dishonesty: front-running attacks on Blockchain". arXiv cs.CR/1902.05164.

[EOS18] EOS. "EOS.IO Technical White Paper". https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md.

[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/.

[Eth18] Etherscan. "The Ethereum Block Explorer". https://etherscan.io/tokens.

[FK97] U. Feige et al. "Making Games Short". In: STOC '97.

[FST10] D. Freeman et al. "A taxonomy of pairing-friendly elliptic curves". In: *J. Cryptol.* (2010).

[GGP+13] R. Gennaro et al. "Quadratic Span Programs and Succinct NIZKs without PCPs". In: EUROCRYPT '13.

[GM17] J. Groth et al. "Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs". In: CRYPTO '17.

[Goo14] L. Goodman. "Tezos — a self-amending crypto-ledger". https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf.

[Gro06] J. Groth. "Simulation-Sound NIZK Proofs for a Practical Language and Constant Size Group Signatures". In: ASIACRYPT '06.

[GWC19] A. Gabizon et al. "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge". ePrint Report 2019/953.

[HBH+18] D. Hopwood et al. “Zcash Protocol Specification”. URL: https://github.com/zcash/zips/blob/master/protocol/protocol.pdf.

[JSS+16] S. Jain et al. “How to verify computation with a rational network”. arXiv cs.GT/1606.05917.

[KB16] R. Kumaresan et al. “Amortizing Secure Computation with Penalties”. In: CCS ’16.

[KGC+17] H. Kalodner et al. “BlockSci: Design and applications of a blockchain analysis platform”. arXiv cs.CR/1709.02489.

[KGC+18] H. A. Kalodner et al. “Arbitrum: Scalable, private smart contracts”. In: USENIX Security ’18.

[KGM19] G. Kaptchuk et al. “Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers”. In: NDSS ’19.

[KMB15] R. Kumaresan et al. “How to Use Bitcoin to Play Decentralized Poker”. In: CCS ’15.

[KMS+16] A. E. Kosba et al. “Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts”. In: SP ’16.

[LTK+15] L. Luu et al. “Demystifying Incentives in the Consensus Computer”. In: CCS ’15.

[MAB+13] F. McKeen et al. “Innovative instructions and software model for isolated execution”. In: HASP ’13.

[MBK+19] M. Maller et al. “Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updateable Structured Reference Strings”. ePrint Report 2019/099.

[MGG+13] I. Miers et al. “Zerocoin: Anonymous Distributed E-Cash from Bitcoin”. In: SP ’13.

[MPJ+13] S. Meiklejohn et al. “A Fistful of Bitcoins: characterizing payments among men with no names”. In: IMC ’13.

[MRK03] S. Micali et al. “Zero-Knowledge Sets”. In: FOCS ’03.

[MS18] I. Meckler et al. “Coda: Decentralized cryptocurrency at scale”. https://cdn.codaprotocol.com/v2/static/coda-whitepaper-05-10-2018-0.pdf.

[Nak09] S. Nakamoto. “Bitcoin: a peer-to-peer electronic cash system”. URL: http://www.bitcoin.org/bitcoin.pdf.

[NVV18] N. Narula et al. “zkLedger: Privacy-Preserving Auditing for Distributed Ledgers”. In: NSDI ’18.

[PA14] N. Popper et al. “Apparent Theft at Mt. Gox Shakes Bitcoin World”. https://www.nytimes.com/2014/02/25/business/apparent-theft-at-mt-gox-shakes-bitcoin-world.html. Accessed 2018-12-27.

[PB17] J. Poon et al. “Plasma: Scalable Autonomous Smart Contracts”. https://plasma.io/.

[Pro18] T. B. Project. “An Overview of Decentralized Trading of Digital Assets”. https://collaborate.thebkp.com/project/TL/document/9/version/10/. Accessed 2018-12-27.

[RCGJ+17] A. Rai Choudhuri et al. “Fairness in an Unfair World: Fair Multiparty Computation from Public Bulletin Boards”. In: CCS ’17.

[Rei16] C. Reiwießner. “From Smart Contracts to Courts with not so Smart Judges”. https://blog.ethereum.org/2016/02/17/smart-contracts-courts-not-smart-judges/.

[RH11] F. Reid et al. “An Analysis of Anonymity in the Bitcoin System”. In: SocialCom/PASSAT ’11.

[RS13] D. Ron et al. “Quantitative Analysis of the Full Bitcoin Transaction Graph”. In: FC ’13.

[Sah99] A. Sahai. “Non-Malleable Non-Interactive Zero Knowledge and Adaptive Chosen-Ciphertext Security”. In: FOCS ’99.

[SCF+15] F. Schuster et al. “VC3: Trustworthy Data Analytics in the Cloud Using SGX”. In: SP ’15.

[SMZ14] M. Spagnuolo et al. “BitIodine: Extracting Intelligence from the Bitcoin Network”. In: FC ’14.

[TR17] J. Teutsch et al. “A scalable verification solution for blockchains”. https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf.

[Val08] P. Valiant. “Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency”. In: TCC ’08.

[VB15] F. Vogelsteller et al. “ERC-20 Token Standard”. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[VBMW+19] J. Van Bulck et al. “Breaking Virtual Memory Protection and the SGX Ecosystem with Foreshadow”. In: *IEEE Micro* (2019).

[Woo17] G. Wood. “Ethereum: A Secure Decentralised Generalised Transaction Ledger”. http://yellowpaper.io.

[WSR+15] R. Wahby et al. “Efficient RAM and control flow in verifiable outsourced computation”. In: NDSS ’15.

[Zcaa] “ZCash Company”. https://z.cash/.

[Zcab] “ZCash Parameter Generation”. https://z.cash/technology/paramgen.html. Accessed: 2017-09-28.

[ZDB+17] W. Zheng et al. “Opaque: An Oblivious and Encrypted Distributed Analytics Platform”. In: NSDI ’17.

[Zha18] W. Zhao. “Bithumb $31 Million Crypto Exchange Hack: What We Know (And Don’t)”. https://www.coindesk.com/bithumb-exchanges-31-million-hack-know-dont-know. Accessed 2018-12-27.