Zapper: Smart Contracts with Data and Identity Privacy

Samuel Steffen
ETH Zurich
Switzerland
samuel.steffen@inf.ethz.ch

Benjamin Bichsel
ETH Zurich
Switzerland
benjamin.bichsel@inf.ethz.ch

Martin Vechev
ETH Zurich
Switzerland
martin.vechev@inf.ethz.ch

ABSTRACT

Privacy concerns prevent the adoption of smart contracts in sensitive domains incompatible with the public nature of shared ledgers.

We present Zapper, a privacy-focused smart contract system allowing developers to express contracts in an intuitive frontend. Zapper hides not only the identity of its users but also the objects they access—the latter is critical to prevent deanonymization attacks. Specifically, Zapper compiles contracts to an assembly language executed by a non-interactive zero-knowledge processor and hides accessed objects by an oblivious Merkle tree construction.

We implemented Zapper on an idealized ledger and evaluated it on realistic applications, showing that it allows generating new transactions within 22 s and verifying them within 0.03 s (excluding the time for consensus). This performance is in line with the smart contract system ZEXE (Bowe et al., 2020), which offers analogous data and identity privacy guarantees but suffers from multiple shortcomings affecting security and usability.

CCS CONCEPTS

• **Security and privacy** → *Privacy-preserving protocols*; *Cryptography*; *Pseudonymity, anonymity and untraceability*; • **Software and its engineering** → *Domain specific languages.*

KEYWORDS

Privacy; Zero-knowledge proofs; Blockchain; Smart contracts

ACM Reference Format:
Samuel Steffen, Benjamin Bichsel, and Martin Vechev. 2022. Zapper: Smart Contracts with Data and Identity Privacy. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560622

1 INTRODUCTION

Smart contracts allow modifying the state maintained by a shared ledger according to well-defined logic. The most widely used ledgers are permissionless (i.e., open to anyone) and hence allow anyone to observe the state and all state changes. This lack of privacy prevents the deployment of smart contracts in potentially sensitive domains such as payments, voting, and medicine.

Privacy for Cryptocurrencies Privacy issues of shared ledgers are particularly well explored in the context of cryptocurrencies. Bitcoin is notoriously vulnerable to deanonymization attacks, which identify the sender of individual transactions by tracing coins [46]. Even privacy-focused cryptocurrencies like Monero [44] are vulnerable to coin tracing attacks [34, 43].

To prevent privacy leaks, Zerocash [47] (commercially deployed as Zcash [33]) cryptographically shields all sensitive aspects of its transactions. Specifically, it leverages an oblivious Merkle tree construction to hide the sender and receiver of a coin, as well as which coin was transferred.

Privacy for Smart Contracts While Zerocash provides strong privacy guarantees, it is not programmable. In contrast, almost all programmable ledgers expose their users to deanonymization attacks or require strong trust assumptions (see §12). The only exception is ZEXE [14], which reliably prevents deanonymization attacks. Conceptually, ZEXE extends Zerocash to programmable *records* (units of data generalizing the concept of a coin) produced and consumed by transactions. ZEXE inherits the strong privacy guarantees of Zerocash, protecting not only the sender's identity, but also the involved records, their data, and the logic itself.

Shortcomings of ZEXE Unfortunately, ZEXE suffers from multiple shortcomings (see §11 for details). First, its applications are prone to vulnerabilities, to the point where even the original authors missed two attacks (presented in §11) on their motivating example. Further, the prevention of one attack as recommended by the authors requires the smart contract programmer to be intimately familiar with the combination of zero-knowledge proofs and key-private public-key cryptography. Second, ZEXE obstructs modular development of applications, as cooperating contracts must typically be aware of each other to prevent future, malicious contracts from bypassing the logic of existing contracts. Third, deploying a new application on ZEXE requires a setup performed by a trusted party to ensure the contract logic cannot be bypassed. Finally, ZEXE relies on a non-standard and low-level programming model in terms of *predicates*, which is unfamiliar to most developers.

This Work: Privacy-preserving Smart Contracts We present *Zapper*, a novel privacy-focused smart contract system. Zapper allows developers to implement smart contracts in an intuitive, Python-embedded frontend with a standard programming model similar to Ethereum. It provides data and identity privacy by hiding the involved parties, the data, and the objects accessed in a transaction. Additionally, Zapper achieves correctness (transactions respect the contract logic), access control (malicious contracts cannot bypass the logic of other contracts), integrity (transactions cannot be tampered with or replayed), and availability (valid transactions are not rejected), thus preventing the vulnerabilities found in ZEXE applications.

Approach Our key technical insight is to leverage a novel combination of an oblivious Merkle tree construction and a non-interactive zero-knowledge (NIZK) processor. Our oblivious Merkle tree construction hides the accessed objects and relies on techniques from Zerocash [33, 47] and ZEXE [14], adapted to our context and avoiding the vulnerabilities of ZEXE applications.

Our NIZK processor performs provably correct state updates without revealing private information, and is inspired by a separate line of work [7–9]. In contrast to ZEXE, using a NIZK processor avoids requiring a trusted party for deploying new contracts. To execute contracts on this processor, Zapper compiles them to a custom assembly format. Importantly, it sandboxes contracts by limiting their interactions to function calls. This access control facilitates modular development of contracts.

We note that selecting and combining these techniques to form a private, efficient, and secure smart contract system is a challenging task, as evidenced by the shortcomings of ZEXE (see also §11).

Contributions To summarize, our main contributions are:

- Zapper, a system to express private smart contracts (§2),
- a compilation to our custom assembly language (§3),
- a cryptographic construction to maintain and efficiently update the Zapper system state (§4–§6 and Fig. 3) while satisfying key security properties (§7),
- an end-to-end implementation[1] of Zapper (§8), and
- a thorough evaluation demonstrating that Zapper is efficient (§9).

2 OVERVIEW

In this section, we provide an overview of Zapper.

2.1 Running Examples: Coin and Exchange

Fig. 1 shows implementations of a coin (Fig. 1a) and a decentralized coin exchange (DEX; Fig. 1b) in our Python-embedded frontend.

Coin A coin consists of an amount (Lin. 2), a currency (Lin. 3), and an owner identified by an address (Lin. 4, discussed shortly).

Users can issue a *transaction* to call a function of the coin. For example, the `transfer` function (Lin. 17) transfers the coin to a new owner by overwriting the `owner` field (Lin. 19). Here, Lin. 18 rejects (i.e., aborts and reverts) the transaction unless the *sender* (the address of the account used to create the transaction) is the coin's owner. The expression `self` indicates the *receiver object* of the transaction, while `self.me` holds the address of the sender.

Function `split` (Lin. 21) splits the coin into two coins while preserving the total amount. The function `merge` (omitted) merges two coins. Here, Lin. 24 decreases the amount of the original coin by `a`, while Lin. 25 creates a new coin with amount `a` via the constructor `create`. To prevent users from creating coins "out of thin air," the `create` function is marked as *internal* (Lin. 7), meaning that it can only be called from within the `Coin` class. The only non-internal constructor is `mint` (Lin. 13), which creates a *new* currency with a fixed total amount. This function leverages the built-in `fresh()` expression (Lin. 14) to obtain a fresh currency identifier. This identifier is guaranteed (with overwhelming probability) to be unique, preventing the minting of pre-existing currencies.

[1]The source code is publicly available at https://github.com/eth-sri/zapper.

Ownership To ensure that Zapper objects are private, Zapper encrypts them under the public key of their owner, which is stored in a dedicated `owner` field implicitly available in every class (see Lin. 4). Specifically, `owner` holds the public key pk_α (serving as the *address*) of an account α. Only users with access to the corresponding secret key sk_α can read the contents of the object or interact with it.

DexOffer and Object Ownership Class `DexOffer` allows a `maker` to offer an exchange of a given `coin` (Lin. 29) for another coin of a specific amount and currency (Lin. 30).

Fig. 1c visualizes an example usage of `DexOffer`. Initially, Alice (👤) and Bob (👤) own one dollar (1\$) and one euro (1€) coin, respectively. To offer a coin exchange, Alice first creates a shared user account (👤) and distributes its key pair ($\text{sk}_\text{shared}, \text{pk}_\text{shared}$) to anyone she is willing to trade with, including Bob. Then, she creates a `DexOffer` object *dex* by calling `create` (Lin. 34), using her own account as the sender and passing (i) the public key of the shared account, (ii) her coin, and (iii) the expected amount of 1€ to be received in return.

To ensure that Alice cannot spend her coin while the offer still stands, Lin. 40 changes the coin's owner to *dex* by calling `transfer`. To this end, `DexOffer` is annotated as `@has_address`, which indicates that its instances are assigned their own *object account* and can therefore own other objects. Specifically, the address of *dex* is available via `self.address` (see Lin. 40). Note that two public keys are relevant to *dex*: while *dex* is owned by pk_shared 🔒, the 1\$ coin is transferred to *dex* using *dex*'s own key pk_{dex} 🔒 (middle of Fig. 1c).

As `transfer` updates the owner of the 1\$ coin to *dex*, Alice can no longer use her own account to spend it. However, the secret key sk_{dex} of *dex* is stored as part of *dex*, allowing Alice and Bob (who know sk_shared) to access it. To prevent both users from spending the coin, Zapper prohibits object accounts to be used as sender accounts for a transaction. Thus, once the coin is transferred to *dex*, the `require` statements in Lin. 18 and Lin. 22 of `Coin` prevent Alice and Bob from interacting with the coin directly, even though they know the secret key sk_{dex} of the coin's owner. Instead, Alice and Bob must interact with the coin via *dex* as follows.

To accept the offer, Bob calls `accept`, using his own account as the sender and passing his 1€ coin (checked in Lin. 44–45). The function transfers the 1\$ coin to Bob (Lin. 47) and Bob's 1€ coin to Alice (Lin. 51). By default, the sender address `self.me` is inherited in a nested function call: in Lin. 51, the call to `transfer` uses Bob as the sender. However, because the 1\$ coin is owned by *dex*, the call in Lin. 47 sets the `sender_is_self` flag (a reserved argument implicitly defined for any function), which sets the sender address inside `transfer` to *dex*. Finally, Lin. 52 in `accept` destroys *dex* and makes it inaccessible to future transactions.

If Bob refuses to accept the offer, Alice can reclaim her 1\$ coin owned by *dex* by calling `abort` (omitted). Further, to prevent unexpected privacy leaks, the owner of an object with its own address (`@has_address`) cannot be modified after construction (see §7).

Discussion The decentralized exchange application implemented in Fig. 1 is inspired by [14, §V] and captures an important practical use case. In particular, `DexOffer` allows exchanging coins without handing over custody of coins to trusted centralized exchanges, which are notoriously vulnerable to attacks [45, 55]. Further, its privacy properties (discussed shortly) hide the most sensitive aspects of trading patterns (most importantly, the user identity and involved

(a) Coin class

```
class Coin(Contract):
  amount: Uint
  currency: Long
  # owner: Address (declared implicitly)

  @constructor
  @internal
  def create(self, amt: Uint, c: Long, o: Address):
    self.amount = amt; self.currency = c
    self.owner = o

  @constructor
  def mint(self, amt: Uint):
    self.amount = amt; self.currency = self.fresh()
    self.owner = self.me

  def transfer(self, recipient: Address):
    self.require(self.owner == self.me)
    self.owner = recipient

  def split(self, amt: Uint) -> Coin:
    self.require(self.owner == self.me)
    self.require(self.amount >= amt)
    self.amount -= amt
    return self.new_obj(Coin.create, amt,
                        self.currency, self.me)
```

(b) DexOffer class

```
@has_address
class DexOffer(Contract):
  maker: Address; coin: Coin
  for_amount: Uint; for_currency: Long
  # owner: Address (declared implicitly)

  @constructor
  def create(self, shared: Address,
             coin: Coin, a: Uint, c: Long):
    self.maker = self.me; self.coin = coin
    self.for_amount = a
    self.for_currency = c
    self.owner = shared
    coin.transfer(self.address)

  def accept(self, other: Coin):
    self.require(
      other.amount   == self.for_amount and
      other.currency == self.for_currency
    )
    self.coin.transfer(
      self.me,
      sender_is_self=True
    )
    other.transfer(self.maker)
    self.kill()
```

(c) Example usage

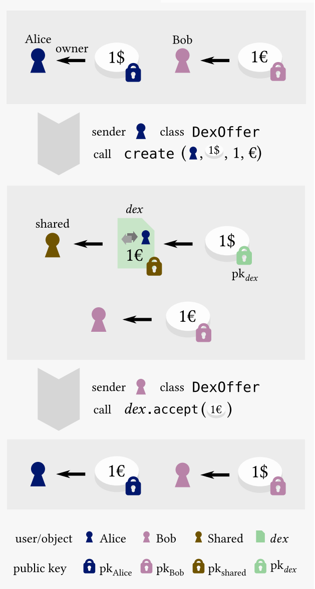


Figure 1: Zapper classes modeling a coin (a) and a decentralized exchange (b), inspired by [14, §V], including an example usage (c).

amounts), thus preventing attackers from exploiting these [24]. See [14] for a more elaborate discussion.

While in our example, the maker shares the offer details with all potential traders, the application can also be instantiated differently to provide more privacy. For example, the maker could share $\text{sk}_{\text{shared}}$ only with a centralized order book service which then connects potential trading partners. Further, the maker can remain anonymous by first creating an ephemeral user account and then transferring the offered coin to this account before creating *dex*. Once *dex* is accepted, the maker can privately transfer the received coin back to its original account.

2.2 Security Properties Guaranteed by Zapper

We now discuss the security properties ensured by Zapper.

Privacy Zapper ensures *identity privacy*: For every transaction, it hides the sender address (i.e., the value of `self.me`). This avoids revealing the `Coin` owner and hides the trading patterns of users.

Zapper also ensures *data privacy*: First, the values of all object fields are only visible to users knowing the owner's secret key. For example, only these users can see a coin's amount, and only users with access to $\text{sk}_{\text{shared}}$ can see the details of *dex*. Second, function arguments and return values of a transaction are only visible to the user creating the transaction. Importantly, this includes the identity of the receiver object (i.e., the value of the `self` argument). In our example, this ensures that coins cannot be tracked: it is hidden which of the coins in the system is modified by a `transfer` transaction. This is critical because revealing the receiver object enables tracing attacks which can compromise the sender's identity [43].

Correctness Zapper ensures that the logic defined in function bodies cannot be violated at runtime. Combined with access control (discussed next), this ensures that the behavior of a coin is completely defined by its implementation, even if used by untrusted users or code. For example, `Coin` ensures that the total amount of all coins of a specific `currency` remains constant after minting.

Access Control Zapper classes are subject to access control, which ensures that correctness cannot be violated by other, potentially malicious, classes. First, as discussed in §2.1, calls to internal functions are only permitted from within the same class. Second, the fields of an object can only be written from within a function of the same class, forcing any state changes to be performed via the function interface. For example, `DexOffer` cannot directly update the `owner` of a coin but must use the `transfer` function instead.

Integrity and Availability Finally, Zapper ensures that transactions cannot be tampered with or replayed (integrity), and that valid transactions are not rejected (availability). In our example, availability ensures that the recipient of a coin is guaranteed to have access to it after `transfer` has been executed.

2.3 Zapper Components

We now discuss the two main components of Zapper. The *Zapper client* allows users to create classes and transactions. The *Zapper executor* stores classes and objects, and executes transactions.

Ledger While users run the Zapper client on their local machine, the Zapper executor runs on top of a shared ledger, whose consensus mechanism maintains a globally consistent view of the system state. The realization of such a ledger is out of the scope of this work—Zapper is ledger-agnostic and could for instance be deployed on an extension of the Zcash [33] ledger, which maintains data structures similar to Zapper.

Assembly Code Before submitting a new Zapper class to the Zapper executor, the Zapper client compiles it to a custom Zapper

```
1  class Coin:
2    amount: Uint
3    currency: Long
4
5    def transfer(self, arg: Address) -> None
6      CID tmp0 self               # tmp0 ⟵ class id of self
7      EQ tmp1 tmp0 'Coin'         # tmp1 ⟵ tmp0 == 'Coin'
8      REQ tmp1                    # reject unless tmp1 == 1
9      LOAD tmp2 self 'owner'      # tmp2 ⟵ self.owner
10     EQ tmp3 tmp2 me             # tmp3 ⟵ tmp2 == me
11     REQ tmp3                    # reject unless tmp3 == 1
12     STORE arg self 'owner'      # self.owner ⟵ arg
13
14   ... # other functions
```

```
17 def transfer(self, recipient: Address):
18   self.require(self.owner == self.me)
19   self.owner = recipient
```

Figure 2: Zasm representation of `Coin`, using named registers for the sender (`me`), receiver (`self`), temporaries (`tmpx`), and arguments (`arg`).

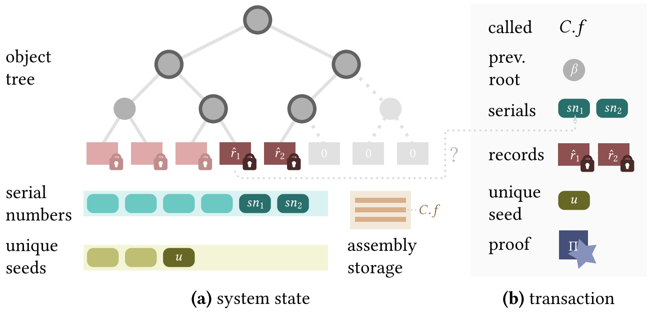


Figure 3: State maintained by the Zapper executor. New transactions insert the darker rectangles and update the rounded circles.

assembly language (Zasm) format. Zasm code consists of instructions for a (virtual) processor ran when executing a transaction.

For example, Fig. 2 shows the Zasm code of `Coin`, focusing on function `transfer`. Lin. 6–8 perform a type check of `self`. This step is necessary to prevent malicious users or code from passing a receiver of non-`Coin` type to `transfer` and thereby bypassing Zapper's access control. Here, **`CID`** loads the *class id* (a number identifying the class) of `self` into a temporary register `tmp0`. The **`EQ`** instruction stores 1 into `tmp1` if and only if the two arguments are equal (0 otherwise), where `'Coin'` represents the class id of `Coin` in a readable fashion. Next, Lin. 9–11 ensure that only the coin owner can transfer coins (see Lin. 18 in Fig. 1a), where `'owner'` represents the numerical offset of field `owner` in `Coin`. Finally, Lin. 12 updates the coin's owner.

Assembly Storage The Zapper executor stores Zasm code in an *assembly storage*. It enforces access control by inferring the type of registers in Zasm code and then checking whether the code respects the relevant access control policies. For instance, Zapper checks that internal functions are only called from within the same class and that only fields of the same class as `self` are written by **`STORE`**.

In order to verify whether potentially untrusted Zasm code matches a trusted Zapper Python class, users can separately compile the class and check equality of Zasm code (analogously to how EVM bytecode is checked to match Solidity code in Ethereum).

System State Fig. 3a visualizes the system state maintained by the Zapper executor. In particular, objects are stored in an *object tree T*. More precisely, the data of an object is encrypted under the owner's public key to obtain a *record*. The records representing the current and past states of any object in the system are stored as leaves in T, which is an append-only Merkle hash tree [40] whose root β is a cryptographic summary of the object states.

Zapper further maintains two auxiliary data structures. First, to invalidate records accessed by transactions (see shortly), Zapper uses an append-only list of unique *serial numbers*. Second, an append-only list of *unique seeds* allows various components of Zapper to produce provably unique values. For instance, these seeds are used to compute values returned by `fresh()` (Lin. 14 in Fig. 1a).

Transactions The Zapper client allows users to execute a function f of a previously registered Zasm class C by sending a transaction to the Zapper executor (see Fig. 3b). Conceptually, this transaction executes the Zasm instructions of $C.f$ on the Zapper processor and updates the state of the involved objects by inserting new records into the object tree. To invalidate the previous state of the objects accessed (by reads or writes) in the transaction, the transaction includes a list of serial numbers which uniquely but privately indicate the accessed records. Enforcing the uniqueness of serial numbers then ensures that each record is accessed at most once. Similarly, a unique seed is produced uniformly at random.

Importantly, the unique seed, serial numbers, and new records do not leak any information about the data, objects, or users involved in the transaction, thus maintaining both data and identity privacy.

Ensuring Correctness The user also includes a NIZK proof Π in the transaction to certify that the new records and serial numbers were computed correctly. This proof is verified by the Zapper executor, which upon success inserts the records, serial numbers, and unique seed into the object tree, serial number list, and unique seed list, respectively (Fig. 3).

3 ASSEMBLY CODE

Next, we provide details on the Zasm code generated by the Zapper client (§3.1). Then, we discuss how this code is processed by the Zapper executor before storing it in the assembly storage (§3.2).

Overall, Zasm allows to enforce access control via type checking and is designed to enable efficient execution on a NIZK processor.

3.1 Assembly Code in Client

The Zapper client compiles classes expressed in Zapper's Python frontend to Zasm code. As this step is conceptually straightforward, we only discuss the resulting Zasm code.

Types and Values Zasm code contains type information. Supported types include unsigned integers (**`Uint`**), addresses (**`Address`**), and pointers to objects. For technical reasons (see Footnote 3 in §6), values produced by `fresh()` are of the special large unsigned integer type **`Long`** precluded from arithmetic operations.

All values are elements of a prime field $\mathbb{F}_q = \{0, \ldots, q-1\}$ for a large prime number q, allowing for efficient correctness checks of processor execution (see §8). Pointers to an object hold the object's *object id*—a unique identifier in $\mathbb{F}_q$ assigned to each object.

Classes and Instructions The Zasm code of a class defines its fields and functions, including the types of fields, function arguments, and return values (see Fig. 2). Further, function bodies are

Table 1: Zapper-specific Zasm instructions.

`REQ` c	Aborts transaction if `reg[`c`]`$\neq 1$
`LOAD` d *oid* i	Loads i-th field of object with id *oid* into `reg[`d`]`
`STORE` s *oid* i	Stores `reg[`s`]` into the i-th field of object with id *oid*
`CID` d *oid*	Writes the class id of object with id *oid* to `reg[`d`]`
`PK` d *oid*	Writes the public key of object with id *oid* to `reg[`d`]`
`NEW` d *cid*	Creates a new object with class id *cid* and writes its object id to `reg[`d`]`
`KILL` *oid*	Destroys the object with id *oid*
`FRESH` d	Writes a unique secret value to `reg[`d`]`
`NOW` d	Writes the current timestamp to `reg[`d`]`

represented by a sequence of Zasm instructions. In Tab. 1, we list the key Zasm instructions—due to lack of space, the table omits standard generic instructions (see App.[2] A.1 for the full instruction set). The instruction set has been specifically designed to allow for efficient generation of NIZK correctness proofs (see §6).

Registers Zasm code operates on named registers `reg`, each holding a value in $\mathbb{F}_q$. The sender address (`self.me` in Zapper's frontend), and function arguments are available in dedicated registers.

Basic Operations Zasm supports standard arithmetic ($+, -, \cdot$), comparison ($<, =$), and conditional assignment operations (assigning some value to a register iff a condition is true). Comparison operations return a value 0 or 1 interpreted as false or true, respectively. Zapper can express boolean operations by arithmetic operations (e.g., $a \mathbin{\&\&} b = a \cdot b$). Operation **`REQ`** enforces assertions.

Loading and Storing Object Data Zasm provides object-aware memory instructions to access objects by their object id. While this prohibits advanced techniques such as pointer arithmetic, it enables efficient access of object data within NIZK proof circuits (see §6).

For instance, the **`LOAD`** instruction first finds the object with object id *oid* and then loads the i-th field into the target register `reg[`d`]` (during compilation, each field of a class is assigned a numerical offset). The **`STORE`** instruction works analogously. Zasm further allows accessing object metadata. First, the **`CID`** instruction gets the class id of an object. This is useful to realize runtime type checks: for example, Lin. 6–8 in Fig. 2 ensure that `self` is a `Coin`. Second, **`PK`** returns the object's own address (public key) if it has one.

Creating and Destroying Objects The **`NEW`** instruction stores the object id of a new object of a given class into a target register. Subsequent **`STORE`** instructions can then be used to populate the fields of the new object. Conversely, the **`KILL`** instruction destroys a given object, making it inaccessible for future instructions.

Fresh Values and Timestamps The **`FRESH`** instruction creates and stores a unique secret value into a target register (see `fresh()`).

The Zapper executor maintains a clock at coarse granularity. The **`NOW`** instruction stores the current timestamp into a target register.

Function Calls Function calls are represented by a special **`CALL`** instruction indicating (i) the called function, (ii) the `sender_is_self` flag determining whether the sender is set to `self.address` or inherited, and (iii) the arguments, including the receiver object id.

[2]Available in the extended version of this paper: https://www.sri.inf.ethz.ch/publications/steffen2022zapper

Control Flow To allow for efficient generation of NIZK correctness proofs (§6), Zasm code does not support control flow, jumps, loops, or operations modifying Zasm instructions at runtime (i.e., we assume a Harvard architecture). In particular, Zasm instructions are always executed in the given order.

However, by representing if-then-else using conditional assignments and statically unrolling loops up to an upper bound, most smart contracts can be expressed in Zasm.

3.2 Assembly Code in Executor

When the Zapper client registers a new Zasm class at the Zapper executor, the latter ensures it does not violate access policies and prepares it for execution on the Zapper processor.

Malicious Code Zasm code received at the Zapper executor cannot be trusted, since an attacker could try to craft malicious Zasm code bypassing the logic specified in existing Zasm classes. For instance, an attacker could use **`STORE`** to increase the `amount` of a `Coin` object from a different class. To prevent such attacks, Zapper enforces multiple access policies, as discussed next.

Access Control Zapper enforces several access policies by default. First, **`STORE`** instructions within a class C can only target objects of type C, ensuring that fields of other classes cannot be written directly. Similarly, to prevent the destruction of unrelated objects, **`KILL`** may only destroy objects of type C. Further, **`NEW`** may only create objects of type C (objects of different type C' must be created via a **`CALL`** to a constructor of C'). Also, **`STORE`** cannot be used to update the owner field of classes annotated as `@has_address` outside constructors (we discuss the necessity of this rule in §7). Finally, the dedicated register holding the sender address (authenticated by Zapper) must not be overwritten by any instruction.

We note that by design, Zapper ensures that the user creating a transaction knows the secret key of the owner of any accessed object (by **`LOAD`**, **`STORE`**, **`CID`**, **`PK`**, or **`KILL`**), as their records must be decrypted. This is *not* equivalent to `self.require(self.owner == self.me)`: a user can only use one sender account (which must be a user account) but may have access to the secret keys of many accounts (including object accounts). For example, in the second transaction of Fig. 1c, Bob uses $\mathrm{sk}_{\mathrm{shared}}$, sk_{dex}, and $\mathrm{sk}_{\mathrm{Bob}}$ to decrypt the inputs, but $\mathrm{pk}_{\mathrm{Bob}}$ as the sender address.

In addition to the default access policies, Zapper allows specifying custom policies for functions. In particular, the annotation `@only(`C`)` declares a function *internal to* a class C, making it inaccessible to any class $C' \neq C$. Zapper also provides an annotation `@internal` as a shorthand for `@only(`C`)`, where C is the current class.

Static Checks To enforce the above policies, the Zapper executor performs a static type analysis on the received Zasm code. First, it checks whether all fields of new objects are initialized (uninitialized objects can violate type safety). Second, it checks whether the code is well-typed (e.g., arithmetic operations are only performed on **`Uint`** values, the types of arguments in **`CALL`** match the target's function signature, etc.) and determines the target class of each **`LOAD`**, **`STORE`**, and **`CALL`** instruction. Next, Zapper checks whether the code satisfies all default and custom access policies described above. If any of these checks fail, the Zasm code is rejected.

Runtime Checks The types of any pointer arguments cannot be checked statically as their value is selected by a potentially malicious user at runtime. Hence, the Zapper executor inserts dynamic type checks for these arguments, relying on the **`CID`** instruction.

For example, consider the `transfer` function of `Coin` (Fig. 1a). To ensure that the first argument `self` is of the expected type `Coin`, Zapper inserts Lin. 6–8 in Fig. 2.

Inlining Calls The Zapper processor does not natively support **`CALL`** instructions. Instead, the Zapper executor inlines the function body of any called function. Here, the sender address register of the called function is correctly inherited or initialized with the caller address, depending on the `sender_is_self` flag. For inlining to succeed, Zapper disallows (mutually) recursive function calls.

Allocating Registers The Zapper processor only supports a fixed number N_{regs} of registers `reg[0]`, . . . , `reg[`N_{regs} `− 1]` for a system parameter N_{regs}. Hence, in a final step, the Zapper executor allocates all named registers to these indexed registers. Then, the Zasm code is stored in the assembly storage, ready to be used by transactions.

4 CRYPTOGRAPHIC COMPONENTS

We next discuss the cryptographic components used in Zapper.

Encryption To encrypt records, Zapper uses a *key-private* [6] asymmetric encryption scheme. This ensures that attackers cannot determine which public key was used to produce a given ciphertext, thus hiding the identity of an encrypted record's owner. We write $\text{Enc}(p, \text{pk}_\alpha, R)$ to denote the encryption of plaintext p under key pk_α with encryption randomness R.

Merkle Tree Zapper relies on a Merkle hash tree [40], which uses a collision-resistant hash function H to derive a root hash β of all records stored as leaves in the object tree T. This allows proving that a given record $\hat{r}$ is in T with root β using a *Merkle certificate* π [38, §2.1.1], and updating β upon insertion of new records.

Hash Functions Zapper relies on a family of cryptographic hash functions $H_i : \{0,1\}^* \to \{0,1\}^{\Omega(\lambda)}$ with security parameter λ and the following properties: (i) H_i is collision-resistant, and (ii) function $f : \{0,1\}^* \times \{0,1\}^{\Omega(\lambda)} \to \{0,1\}^{\Omega(\lambda)}$ defined as $f(x,k) := H_i(x \parallel k)$ is a pseudorandom function (i.e., for uniformly random k, the function $f(\,\cdot\,, k)$ is indistinguishable from a random function).

Various components of Zapper rely on H_i to derive a unique secret value z using the following generic construction, inspired by ZEXE [14] and Zcash [33]. For unique U and private R,

$$z = H_i(U \parallel R). \tag{1}$$

Computed as in Eq. (1), z has two key properties:

- *Secrecy:* For uniformly random R, any user not knowing R cannot distinguish z from a uniformly random value. This follows from pseudorandomness.
- *Uniqueness:* with overwhelming probability, z is unique, even for adversarially chosen R. This follows from collision-resistance.

NIZK Proofs For a predicate ϕ and *public input* x, a NIZK proof [12, 26] allows a prover to demonstrate that it knows a *private input* w s.t. $\phi(x; w)$ holds, without leaking any information about w beyond the fact that ϕ holds. In this work, x includes the executed function body and information on the state before and after execution, while w includes private information known to the sender (e.g., secret keys), allowing ϕ to check that the function was executed correctly.

Zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) [11, 29] are NIZK proof constructions allowing efficient proof generation and verification for any arithmetic *proof circuit* ϕ. To ensure correctness, Zapper relies on an SE-SNARK [29]: a zk-SNARK satisfying perfect zero-knowledge, perfect completeness, and simulation-extractability [29].

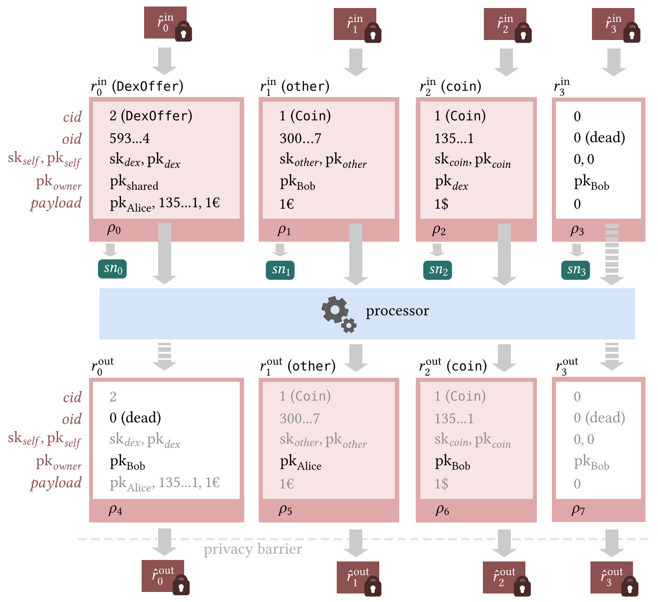


Figure 4: Visualization of a transaction for `DexOffer.accept` (Fig. 1c).

5 TRANSACTIONS

We now discuss how the Zapper system state is represented and updated by transactions. Most importantly, our approach hides the accessed objects by an oblivious Merkle tree construction. To this end, we rely on techniques from Zerocash [33, 47] and ZEXE [14] but adapt them to our context and avoid the vulnerabilities of ZEXE.

5.1 Example Transaction

To provide some intuition, we first discuss an example transaction calling the `accept` function of `DexOffer` (Fig. 1b).

Input Records The `accept` function accesses several objects: (i) the `DexOffer` object `self`, (ii) the coin `other` transferred to the `maker`; and (iii) the coin `coin` transferred to the transaction sender.

Fig. 4 shows how `accept` modifies these objects when called by Bob (sender pk_{Bob}) with arguments *args* = (593...4, 300...7) indicating the object ids of `self` and `other` (see the second transaction in Fig. 1c). At a high level, Zapper first loads the *encrypted records* $\hat{r}_0^{\text{in}}, \ldots, \hat{r}_2^{\text{in}}$ of the accessed objects, which includes $\hat{r}_2^{\text{in}}$ containing the state of `coin`, from the object tree (we discuss $\hat{r}_3^{\text{in}}$ shortly). Next, these are decrypted to obtain *plain records* r_i^{in}.

Plain Records Specifically, the *plain record* of an object is a tuple

$$r = (cid, oid, \text{sk}_{self}, \text{pk}_{self}, \text{pk}_{owner}, payload),$$

where *cid* and *oid* are the class id and object id, respectively, pk_{owner} is the public key of the owner, and *payload* holds the values of the remaining fields. Further, $(\text{sk}_{self}, \text{pk}_{self})$ is the key pair of the object account. For simplicity, we also assign an account to objects not annotated as `@has_address`, however, this account is never used. Note that in general, pk_{self} (belonging to the object) and pk_{owner} (belonging to the object's owner) are keys of different accounts.

Encrypted Records Before encryption, a plain record r is extended by a secret *serial nonce* ρ (a globally unique number later used to invalidate outdated records). Then, it is encrypted under the public key of the object's owner to yield the *encrypted record*

$$\hat{r} = \text{Enc}((r, \rho), r.\text{pk}_{owner}, R),$$

where R is some encryption randomness. Note that in contrast to ZEXE, Zapper relies on encryption instead of commitments, preventing the denial-of-funds attack in ZEXE (see §11).

Processor After decryption, the plain records are fed to the Zapper processor, which executes the Zasm code of `accept` to produce the plain output records r_i^{out} (Fig. 4 highlights changed entries).

Output Records Finally, the plain records r_i^{out} are extended by fresh serial nonces ρ_i and encrypted under the owners' public keys to obtain $\hat{r}_i^{\text{out}}$. These records will be published to the ledger and inserted into the object tree once the transaction is accepted.

Dead Records When an object is destroyed by **KILL**, the processor sets the object id *oid* of the corresponding record to the reserved value 0, indicating that this record is *dead*. Further, pk_{owner} of the record is set to the sender's public key such that the resulting record is hidden from other users. This is important as dead records may include stale data of the previously represented object.

In Fig. 4, the `DexOffer` object $\hat{r}_0^{\text{in}}$ is destroyed, hence *oid* of r_0^{out} is set to 0 and pk_{owner} is set to pk_{Bob}. The payload component of r_0^{out} still contains private data (e.g., the value of the `for_amount` field), but as it will be encrypted for Bob, no other user learns this.

The processor accepts exactly N_{obj} plain records as inputs, for a system parameter N_{obj}. The inputs are appropriately padded using artificial dead records. Similarly, the processor returns N_{obj} output records. In Fig. 4, it is $N_{\text{obj}} = 4$ and $\hat{r}_3^{\text{in}}$ is used as a padding record.

Transaction Contents Importantly, the steps visualized in Fig. 4 are hidden from the Zapper executor to maintain data and identity privacy (see the indicated privacy barrier). The data published by the Zapper client only consists of the encrypted output records $[\hat{r}^{\text{out}}]$ (bottom row in Fig. 4; we use the notation $[\cdot]$ to indicate a list) and some bookkeeping data (see also Fig. 3b).

Formally, a transaction $tx = (C.f, \beta, [sn], [\hat{r}^{\text{out}}], u, \Pi)$ consists of the fully qualified function name $C.f$ of the called function, the current root hash β of T, the serial numbers $[sn]$ of accessed records, a list $[\hat{r}^{\text{out}}]$ of new records, a unique seed u, and a NIZK proof Π certifying correctness. §5.2–§5.3 explain the purpose of these items.

5.2 Creating Transactions

Next, we describe how the Zapper client creates a transaction tx for a user who wishes to call a function $C.f$ with arguments *args*.

Simulation The Zapper client first loads the Zasm code *zasm* for $C.f$ from the assembly storage. It then locally simulates the execution of *zasm* with arguments *args*. During simulation, the

Algorithm 1 The main NIZK proof circuit ϕ.

```
1:  Public inputs:
2:      $C.f$, $\beta$, $[sn]$, $[\hat{r}^{\text{out}}]$, $u$ as in Fig. 3b, timestamp $t$, code $zasm$
3:  Private inputs:
4:      $C'.f'$: called class and func. id      $[r^{\text{out}}]$: plain output records
5:      $\text{sk}_{me}, \text{pk}_{me}$: key pair of sender   $[\pi]$: input record Merkle certificates
6:      $args$: arguments for call             $[\text{sk}_\alpha]$: input record secret keys
7:      $[\hat{r}^{\text{in}}]$: encrypted input records   $[R]$, $[R^{\text{pr}}]$: randomness
8:  Auth. sender: $\text{pk}_{me} \overset{!}{=} \text{derivePk}(\text{sk}_{me})$ and $\text{isUser}(\text{pk}_{me}) \overset{!}{=} \text{true}$
9:  Check function: $C.f \overset{!}{=} C'.f'$
10: for $i \in \{0, \ldots, N_{\text{obj}} - 1\}$ do
11:     Decrypt record: $(r_i^{\text{in}}, \rho_i^{\text{in}}) \leftarrow \text{Dec}(\hat{r}_i^{\text{in}}, \text{sk}_{\alpha_i})$
12:     if $r_i^{\text{in}}.oid \neq 0$ then
13:         Check that $\hat{r}_i^{\text{in}}$ is in Merkle tree with root $\beta$ (using $\pi_i$)
14:     else
15:         Check serial nonce: $\rho_i^{\text{in}} \overset{!}{=} H_2(i + N_{\text{obj}} \parallel u \parallel R_{i+N_{\text{obj}}})$
16:     Check serial number: $sn_i \overset{!}{=} H_1(\rho_i^{\text{in}} \parallel \text{sk}_{\alpha_i})$
17: Run processor (Alg. 2) for $zasm$, $u$, $t$, $\text{pk}_{me}$, $args$, $[r^{\text{in}}]$, $[r^{\text{out}}]$, $[R^{\text{pr}}]$
18: for $i \in \{0, \ldots, N_{\text{obj}} - 1\}$ do
19:     Compute serial nonce: $\rho_i^{\text{out}} \leftarrow H_2(i \parallel u \parallel R_i)$
20:     Check encryption: $\hat{r}_i^{\text{out}} \overset{!}{=} \text{Enc}((r_i^{\text{out}}, \rho_i^{\text{out}}), r_i^{\text{out}}.\text{pk}_{owner}, R_{i+2N_{\text{obj}}})$
```

most recent records of any pre-existing objects accessed by *zasm* (due to **LOAD**, **STORE**, **CID**, **PK**, or **KILL**) are fetched from a local copy of the object tree, collected in a list $[\hat{r}^{\text{in}}]$, and decrypted using the owners' secret keys. As a result, Zapper obtains (i) the return value of $C.f$, which is returned to the user; and (ii) the list $[r^{\text{out}}]$ of plain output records, which represent the updated states of the objects in $[\hat{r}^{\text{in}}]$ and any new objects. The list $[\hat{r}^{\text{in}}]$ is then padded by an appropriate number of dead records.

Correctness Proof Circuit To prove that $C.f$ was executed correctly, the Zapper client creates a NIZK proof Π.

Alg. 1 shows the proof circuit ϕ for Π. Conceptually, the public inputs of ϕ (Lin. 1–2) are provided by the Zapper executor when verifying Π. In contrast, the private inputs of ϕ (Lin. 3–7) are provided by the Zapper client when creating Π. These inputs include private information such as the keys $(\text{sk}_{me}, \text{pk}_{me})$ of the sender.

Sender Authentication To ensure that the sender cannot be impersonated by an unauthorized user, we check that the user creating Π knows the secret key of the sender account. More precisely, Lin. 8 uses derivePk to check whether sk_{me} corresponds to pk_{me}.

Checking the Function Both the public and private inputs of ϕ include identifiers of the called class C and function f. By checking that these match (Lin. 9), the proof Π acts as a signature on $C.f$, ensuring that $C.f$ cannot be changed once the proof is generated.

Enforcing Object Ownership Zapper objects can be owned by other objects. For instance, during the lifetime of *dex* in Fig. 1, the coin `coin` is owned by *dex*. The `self.require(self.owner == self.me)` statements in `Coin` ensure that only *dex* can call its functions. Unfortunately, reflecting these requirements in ϕ by checking that pk_{me} equals the coin's owner address is not sufficient to prevent users from directly calling the functions of a coin owned by *dex*:

the secret key sk_{dex} is stored as part of *dex*'s record and users with access to *dex* could hence use $(\mathrm{sk}_{dex}, \mathrm{pk}_{dex})$ as sender account keys.

To prevent such object impersonation attacks, the public keys in the system are partitioned into keys of user and object accounts. Lin. 8 explicitly checks that pk_{me} belongs to a user account. In our implementation (§8), isUser returns the key's least significant bit.

Accessing Objects Lin. 11–13 access the input records $[\hat{r}^{\text{in}}]$ while hiding their location in T. First, Lin. 11 checks that $[\hat{r}^{\text{in}}]$ are correctly decrypted, yielding both serial numbers $[\rho^{\text{in}}]$ (discussed shortly) and plain records $[\hat{r}^{\text{in}}]$. Next, for each non-dead record $\hat{r}_i^{\text{in}}$, Lin. 13 verifies a Merkle certificate π_i (passed as a private input to ϕ) showing that $\hat{r}_i^{\text{in}}$ is a leaf of the current object tree with root β.

Preventing Access of Outdated State Recall that new records are appended to the object tree without replacing their original version (note that replacement would leak the accessed object). Hence, the construction presented so far only ensures that the accessed records represent *some* previous state of the respective objects, but not necessarily the most recent one. Therefore, we need to privately invalidate outdated records. To this end, Zapper relies on a technique introduced in Zerocash [47], which uses serial numbers to privately identify and invalidate accessed records.

In particular, to invalidate the accessed records $\hat{r}_i^{\text{in}}$, Zapper derives and publishes a unique *serial number* for each $\hat{r}_i^{\text{in}}$ as

$$sn_i = H_1(\rho_i^{\text{in}} \parallel \mathrm{sk}_{\alpha_i}). \tag{2}$$

Here, ρ_i^{in} is the serial nonce contained in $\hat{r}_i^{\text{in}}$ and sk_{α_i} is the owner's secret key used to decrypt $\hat{r}_i^{\text{in}}$. Eq. (2) is checked in Lin. 16.

Because the computation of the serial number (Lin. 16) follows the generic construction in Eq. (1), assuming the serial nonces ρ_i^{in} are globally unique (discussed shortly), serial numbers of different records cannot collide (*uniqueness*). Further, the serial number sn_i is indistinguishable from a uniformly random value for any user not knowing sk_{α_i} (*secrecy*). This is important to ensure privacy: otherwise, a malicious user Eve who created $\hat{r}_i^{\text{in}}$ for owner Alice could tell when Alice accesses it by watching for sn_i.

Overall, the Zapper client computes the serial numbers $[sn]$ and includes them in *tx*. The Zapper executor will ensure that these are globally unique, enforcing that any record can be accessed at most once. As this also applies to records whose state is not changed but only read (e.g., by **LOAD**), the Zapper processor output includes plain records of objects which were only read. This ensures that a fresh record representing these objects is re-inserted into the object tree.

Processor Execution Lin. 17 ensures that running the program *zasm* with arguments *args* and current timestamp t on inputs $[r^{\text{in}}]$ results in the plain output records $[r^{\text{out}}]$. As this step is more involved, we discuss it separately in §6.

Deriving New Serial Nonces Before encrypting r_i^{out}, Zapper derives a new serial nonce ρ_i^{out} for r_i^{out} as in Lin. 19, repeated here:

$$\rho_i^{\text{out}} = H_2(i \parallel u \parallel R_i). \tag{3}$$

Here, u is a public and globally unique seed, and R_i is fresh randomness. Similarly, the serial nonce ρ_i^{in} of any dead *input* padding record is computed as in Lin. 15, repeated here:

$$\rho_i^{\text{in}} = H_2(i + N_{\text{obj}} \parallel u \parallel R_{i+N_{\text{obj}}}). \tag{4}$$

The seed u is selected uniformly at random by the Zapper client and included in the transaction *tx*. Like the serial numbers, u is enforced to be globally unique by the Zapper executor (note that for a large seed space, the selected u is unique with overwhelming probability). As we will see in §6, the seed u is also used by the processor to derive other unique values. Due to the uniqueness property of the construction in Eq. (1), the nonces computed by Eqs. (3)–(4) are globally unique with overwhelming probability.

In ZEXE, the serial numbers of dead inputs (called "dummy" by the authors) are selected freely by the user, allowing a "lock-out" attack on applications with shared keys (see §11). In contrast, Zapper prevents this attack using the construction in Eq. (4).

Encryption and Proof Generation Finally, Zapper encrypts the plain records $[r^{\text{out}}]$ along with their serial nonces using the respective owners' public keys to obtain the output records $[\hat{r}^{\text{out}}]$ (see Lin. 20). To complete the data in *tx*, the Zapper client generates a NIZK proof Π for ϕ. The transaction *tx* is then sent to the ledger.

5.3 Processing Transactions

We next describe how the Zapper executor processes transactions.

Validity Checks When the Zapper executor receives a transaction $tx = (C.f, \beta, t, [sn], [\hat{r}^{\text{out}}], u, \Pi)$, it first looks up the assembly code *zasm* for the called function $C.f$ in the assembly storage and prepares the public inputs $C.f, \beta, [sn], [\hat{r}^{\text{out}}], u, t, zasm$ for the proof circuit ϕ, where t is the current timestamp. Then, the ledger

(i) checks the validity of the proof Π;
(ii) checks if β is a valid previous root hash of T;
(iii) checks if the serial numbers in $[sn]$ are unique and do not already occur in the serial number list; and
(iv) checks if u does not already occur in the unique seed list.

State Updates If any of the validity checks fail, the transaction is rejected. Otherwise, the system state is updated as follows:

(1) For all $sn_i \in [sn]$, insert sn_i into the serial number list.
(2) Insert u into the unique seed list.
(3) For all $\hat{r}_i^{\text{out}} \in [\hat{r}^{\text{out}}]$, insert $\hat{r}_i^{\text{out}}$ into the object tree T.

Concurrent Transactions Note that another transaction tx' may have been accepted since the Zapper client started creating *tx*. For this reason, in the validity check (ii) above, β is not required to be the most recent root hash of T, but may be any previous root hash. As long as the unique seed and records accessed in *tx* are distinct from the unique seed and records accessed in any previous tx', their ordering does not matter and *tx* remains valid. In other words, concurrent transactions accessing *distinct* objects will not affect each other. Importantly, a third party which does not have access to (i.e., does not know the owner secret keys of) any object involved in another transaction *tx* cannot perform a front-running attack and block *tx* by trying to consume the same object(s).

If two concurrent transactions access the same records (i.e., read or write the same object), the ledger rejects the transaction *tx* it receives last. Assuming all assertions still hold and the involved objects still exist, the user creating *tx* can always re-create it.

We assume that the granularity of timestamps is coarse enough (e.g., in the order of hours) to account for the delay between transaction generation and verification. In case the current timestamp changes in-between, the user must re-create the transaction.

Algorithm 2 The sub-circuit checking Zasm code execution.

```
1: Inputs: $zasm$, $u$, $t$, $\mathrm{pk}_{me}$, $args$, $[r^{\mathrm{in}}]$, $[r^{\mathrm{out}}]$, $[R^{\mathrm{pr}}]$ as in Alg. 1
2: for $i \in \{0, \ldots, N_{\mathrm{fresh}} - 1\}$ do
3:     $oid_i \leftarrow H_3(i \parallel u \parallel R^{\mathrm{pr}}_{3i})$; $f_i \leftarrow H_5(i \parallel u \parallel R^{\mathrm{pr}}_{3i+2})$
4:     $\mathrm{sk}_i \leftarrow H_4(i \parallel u \parallel R^{\mathrm{pr}}_{3i+1})$; assert $\neg$isUser(derivePk($\mathrm{sk}_i$))
5: $state_0 \leftarrow ([r^{\mathrm{in}}], (\mathrm{pk}_{me}, args, 0, \ldots, 0), [oid], [\mathrm{sk}], [f])$
6: for $i \in \{0, \ldots, N_{\mathrm{cycles}} - 1\}$ do
7:     $state_{i+1} \leftarrow$ evalInst($zasm_i$, $state_i$)
8: check output state: $state_{N_{\mathrm{cycles}}} \overset{!}{=} ([r^{\mathrm{out}}], \cdot, \cdot, \cdot, \cdot)$
```

6 PROVING PROCESSOR EXECUTION

We now discuss our zero-knowledge processor, which allows the Zapper client to provably execute Zasm code. Our processor is inspired by previous work [7, 8], but adapted to reduce the cost of the most expensive operations: we prefetch all accessed objects (see also §5) and precompute the result of cryptographic operations.

6.1 Emulating the Processor

In Alg. 2, we show the sub-circuit checking correct execution of the provided Zasm code. This sub-circuit is part of the main proof circuit ϕ (Alg. 1) and emulates the cycles of the Zapper processor.

Precomputation Lin. 2–4 precompute values useful during the execution of the processor and will be discussed later.

State The processor state $state = ([r], regs, [oid], [\mathrm{sk}], [f])$ consists of the current plain records $[r]$ of the N_{obj} involved objects (some of which may be dead), the array $regs$ of N_{regs} register values, and three lists $[oid]$, $[\mathrm{sk}]$, $[f]$ of precomputed values. Lin. 5 initializes this state, where the sender address pk_{me} is (by convention) placed in the first register, followed by $args$ and zero padding.

Cycles Lin. 7 uses the sub-circuit evalInst($zasm_i$, $state_i$) to capture how the processor executes a single instruction $zasm_i$ on input state $state_i$. Similar to [7], as proof circuits do not allow for control flow, evalInst evaluates the result of *each* possible instruction in parallel and then selects the correct result to update $state_i$ according to the instruction $zasm_i$ using a multiplexer. To reduce the proof circuit size, many parts of the computation (e.g., register and object accesses; see shortly) are shared amongst instructions.

Analogously to [7], Alg. 2 unrolls the processor cycles by chaining N_{cycles} copies of evalInst for a system parameter N_{cycles}, supporting any Zasm program with at most N_{cycles} instructions (programs with less than N_{cycles} instructions are padded with no-op instructions). As a consequence, any function of a Zapper class may consist of at most N_{cycles} instructions. However, we note that such limits on the execution length are already commonly used in existing smart contract systems (e.g., Ethereum's block gas limit).

Final State Finally, Lin. 8 checks whether the plain records in the final state $state_{N_{\mathrm{cycles}}}$ match the expected plain records $[r^{\mathrm{out}}]$.

6.2 Evaluating Instructions

We next discuss how evalInst evaluates a single processor cycle.

Register Access To load values from or store values to a register, we use a linear loop to select the target register based on its index. As N_{regs} is small in practice, this step is relatively efficient.

Basic Operations Arithmetic operations work on the *regs* list and mostly correspond to the native field operations of the proof circuit. However, we impose additional checks to prevent unintended over- or underflows of field elements where necessary. For example, native addition or subtraction inside the proof circuit wraps at the field prime ($\approx 2^{255}$ in our implementation, see §8), which may be unexpected. We hence restrict values of type **Uint** to be in $[0, 2^{120})$ and reject any operation leading to a result outside this range. Values of type **Long** are not restricted, but cannot (by the type system) be used in arithmetic operations. [3]

Loading and Storing Object Data To implement **LOAD**, we perform a linear lookup over the plain records $[r]$ to find the targeted object id and field. We proceed analogously for **STORE**, **CID**, and **PK**.

Smart contracts typically only access few objects in a transaction. Therefore, the number of objects N_{obj} in $[r]$ can be set to a small constant in practice, making the above lookup relatively efficient.

As in [8], the memory of the Zapper processor is stored in a Merkle tree. However, by prefetching the memory of *few* input objects in advance (Alg. 1), Zapper induces significantly less overhead than checking a Merkle tree memory access in *each* processor cycle.

Destroying Objects For the **KILL** instruction, the circuit simply marks the targeted record as dead by setting its *oid* component to 0 and its owner public key pk_{owner} to pk_{me}. Analogously to **STORE**, the targeted record is found using a linear lookup.

Creating Objects The **NEW** instruction creates an object by initializing a dead record with a new object id and secret key. Following the generic construction in Eq. (1), the object id oid_i of the i-th new object created in *zasm* is derived as shown in Lin. 3. Here, u is the unique seed for the current transaction, and R is a value chosen uniformly at random and provided to ϕ as a private input. Lin. 3 ensures that oid_i is globally unique, even if R is selected by a malicious user (*uniqueness*). This is important, as creating a new object o whose object id matches a pre-existing object o' would allow hijacking o' and violate correctness. Further, *oid* is hidden from all other users (*secrecy*), thereby ensuring data privacy.

Analogously, the secret key sk_i of the i-th new object is derived as shown in Lin. 4. Here, to ensure that sk_i indeed corresponds to an object account, the Zapper client repeatedly samples uniform randomness R^{pr}_{3i+1} until the assertion in Lin. 4 holds.

Fresh Values The **FRESH** instruction computes a secret and unique value f. The i-th such unique value f_i is computed as shown in Lin. 4 (following Eq. (1)), for unique seed u and uniformly random R.

Precomputation The computations of oid_i, sk_i, and f_i are based on cryptographic hash functions H_i, which are relatively expensive to evaluate within the proof circuit. Hence, instead of computing these in *each* processor cycle, we precompute a fixed number N_{fresh} of these values in advance (see Lin. 2–4 in Alg. 2). Whenever such a value is required in *zasm*, we select the next unused value, assuming a Zapper function requires at most N_{fresh} such values.

Discussion: Universality Generally, zk-SNARKs require a trusted setup, which either depends on the proof circuit (*non-universal* schemes) or not (*universal* schemes). By emulating processor execution, Zapper can use the same proof circuit to verify execution of

[3] The 120 bits of a **Uint** value are not sufficient to provide collision-resistance of values produced by **FRESH**. Hence, we use a separate data type **Long** for these values

arbitrary Zasm programs (respecting the relevant bounds such as N_{cycles}). This allows Zapper to be instantiated with an efficient non-universal SE-SNARK scheme such as GM17 [29] (see §8), without requiring a trusted setup per program.

An alternative design could use a universal scheme such as Marlin [18] to dynamically build a separate proof circuit per Zasm program, again avoiding a trusted setup per program. Unfortunately, Marlin is significantly more expensive than GM17: proof generation for a circuit of the same size (2 million R1CS constraints, see §9.3) takes 17.1 s using GM17 and 116.7 s using Marlin. While using Marlin would avoid processor emulation overhead, we expect this does not compensate for its higher cost: Zapper's proof circuit size is dominated by cryptographic operations, most of which cannot be significantly reduced (see Fig. 5a and §9.2). Still, instantiating Zapper with a universal zk-SNARK presents an interesting trade-off: Despite lower performance, such a system should allow easier extension to more complex Zasm instructions without requiring a new trusted setup for every extension.

7 SECURITY PROPERTIES

In this section, we discuss the privacy, correctness, integrity, and availability properties ensured by Zapper. We note that by construction, Zapper also ensures access control as defined in §3.2.

Attacker Model We consider an active adversary which statically corrupts a set of users and can intercept transactions by honest users. It can craft arbitrary transactions from scratch or by modifying intercepted transactions. See App.^{2} A.2 for a formal definition.

7.1 Privacy

Zapper achieves both data and identity privacy. In particular, only users with access to the secret key of an object's owner can observe when and how the object is created, read, modified, or destroyed.

Ideal World To formalize our notion of privacy, we introduce an ideal world specifying the information available to each user. We sketch this ideal world below (see App.^{2} A.2 for a formal definition).

The ideal world maintains the plaintext state of all objects and allows users to make function calls, which are executed according to the semantics of Zasm. When a user executes a function $C.f$ with arguments *args*, other users of the system only learn the following:

- Any user learns that $C.f$ is called and that the conditions of all `REQ` instructions are satisfied.
- All users who *can access* (see below) an object involved in a `LOAD`, `STORE`, `KILL`, `NEW`, `CID`, or `PK` operation learn the object id and the loaded (for `LOAD`) or stored (for `STORE`) value.

In particular, reading and writing fields is possible without revealing the target object's identity to any users without access to the object. Further, the arguments *args*, the return value, and the identity of the sender account are not visible, unless these are explicitly revealed to some other user by `STORE`. The same applies to any intermediate results of arithmetic operations.

Accessible Objects We say that user U *can access* object o if U *knows* the secret key of o's owner. In particular, U *knows* not only its own or shared user secret keys, but also the secret keys of objects o if (i) U *knows* the secret key of o's owner, or (ii) U created o (and thereby, its secret key) but is not necessarily its current owner. For example, in Fig. 1c, Bob knows sk_{Bob}, $\text{sk}_{\text{shared}}$, and sk_{dex} and can hence access *dex* and both coins (1€ and 1\$). See App.^{2} A.2 for details.

Recall that Zapper prohibits changing the owner of objects with an address (see `@has_address`) at runtime. This prevents unexpected privacy leaks: if U owns o which in turn owns o', then changing the owner of o would not invalidate U's access to o', as U would still know the secret key of o.

Privacy Thm. 7.1 informally states our privacy notion.

Theorem 7.1 (Privacy, informal). *From transactions created by honest users, the adversary cannot learn more than in the ideal world.*

In App.^{2} A.4, we formalize and prove Thm. 7.1. At a high level, privacy is ensured by the zero-knowledge property of the SE-SNARK, the key-privacy of the encryption scheme, and the pseudorandomness of H_i (see §4).

Practical Considerations The location or IP address from which a transaction is submitted may leak the identity of the sender in practice. Hence, as in similar systems [47], users may want to submit transactions via an anonymous overlay network such as Tor [22].

7.2 Correctness, Integrity, Availability

We now informally present the remaining security properties (as introduced in §2), which we formalize in App.^{2} A.2–A.3.

Theorem 7.2 (Correctness, informal). *The adversary cannot violate the logic specified in contracts registered at the Zapper executor.*

Theorem 7.3 (Integrity, informal). *Valid transactions cannot be modified "in flight" or replayed by the adversary.*

Theorem 7.4 (Availability, informal). *Honest users can realize valid transactions unless the adversary actively interferes.*

At a high level, correctness is enforced by the construction of ϕ and the soundness property of the SE-SNARK. Further, the non-malleability of SE-SNARKs prevents transactions from being tampered with, and checking the uniqueness of serial numbers prevents replay attacks. Note that since our attacker model allows the adversary to intercept every transaction, it can always block the *current* transaction. However, if the adversary does not interfere with a transaction, Thm. 7.4 states that it will always be accepted. In particular, the adversary cannot "lock" an object owned by an honest user by publishing a colliding serial number or refusing to share the decryption key. This prevents the "Faerie Gold" attack of Zerocash (allowing attackers to permanently block coins of honest owners) [33] and two attacks on ZEXE (discussed in §11).

8 IMPLEMENTATION

We implemented Zapper in an end-to-end system consisting of a Python frontend (3k LoC) exposed to application developers, and a Rust backend (4k LoC) performing cryptographic tasks and relying on the efficient arkworks libraries [1]. Our implementation includes an idealized centralized ledger, which can be replaced by a shared ledger in an actual deployment. Below, we describe how the cryptographic primitives are instantiated in our implementation. All these primitives have been used in existing systems.

Table 2: Evaluated example Zapper applications and classes. We indicate the number of functions (#fun), and the min/max number of Zasm instructions in the functions of each class (inst).

App	Description	Classes	#fun (inst)
Auction	A private decentralized coin auction.	Auction$	3 (33–48)
Coin	A private untraceable coin.	Coin (Fig. 1)	5 (6–29)
Exchange	A private decentralized coin exchange.	DexOffer$ (Fig. 1)	3 (18–36)
Heritage	A heritable coin wallet with anonymous heirs and private shares.	Share Wallet$	2 (7–8) 7 (9–80)
Reviews	A double-blind peer-review system for academic papers.	Review Result Paper	3 (8–25) 3 (10–11) 4 (17–32)
Tickets	A public transport ticketing system with untraceable multi-journey tickets.	TicketProof Ticket$	1 (7) 4 (10–25)
WorkLog	A system for aggregate working hours reports hiding check-in/-out times.	Aggregated WorkLog	1 (7) 4 (9–23)

$ makes use of the Coin class

Proving Scheme and Elliptic Curves We use the simulation-extractable GM17 [29] zk-SNARKs over the pairing-friendly Barreto-Lynn-Scott [5] curve BLS12-381 introduced in Zcash [33]. The constraints of the proof circuit ϕ are then expressed in the scalar field $\mathbb{F}_q$ of BLS12-381, where $q \approx 2^{255}$. To allow for efficiently emulating the Zapper processor in ϕ, Zasm code operates on values in $\mathbb{F}_q$.

Like Zcash, we rely on curve embedding to efficiently evaluate cryptographic primitives (see below) within ϕ. In particular, our primitives use the Jubjub [13] twisted Edwards curve, whose base field matches the scalar field $\mathbb{F}_q$ of BLS12-381. This allows us to natively evaluate Jubjub curve operations in ϕ.

Hash Functions Like ZEXE [14], we rely on Pedersen and Blake2s hashes. In particular, we instantiate the hash function H for the object tree by the Pedersen hash [33, §5.4.1.7] with 4-bit windows over Jubjub. H is collision-resistant assuming it is hard to compute discrete logarithms in Jubjub [41]. As Pedersen hashes do not provide pseudo-randomness, we use the pseudo-random Blake2s hash function [2] to instantiate $H_i(x) := \text{Blake2s}(i \parallel x)$ as per §4.

Encryption Like the Dusk Network [39], we use a hybrid encryption scheme based on Poseidon [28] and ElGamal [23]. In particular, to encrypt a plain record r, we first select a random curve point k on Jubjub and encrypt r with key k using Poseidon [28] in the DuplexSponge framework [10, 36]. Then, we encrypt k using ElGamal [23] over Jubjub with the owner's public key. As ElGamal encryption is key-private [6], this hybrid scheme is also key-private.

9 EVALUATION

We now evaluate our implementation of Zapper (§8), demonstrating that it is highly efficient. All our experiments are conducted on a desktop machine with 32 GB RAM and 12 CPU threads at 3.70 GHz.

9.1 Example Applications

To demonstrate the expressiveness of Zapper, we implemented the 7 applications described in Tab. 2 in Zapper's Python frontend using a total of 12 classes. The applications span a variety of domains and correspond to realistic use cases. The Coin and Exchange apps

Table 3: Evaluation parameters and runtimes.

(a) Values for parameters.

Tree height	N_{height}	32
Objects in tx	N_{obj}	4
Fresh values	N_{fresh}	4
Processor cycles	N_{cycles}	100
Registers	N_{regs}	10
Object fields	N_{fields}	9

(b) Runtimes for example scenarios.

	Step	Time (std. dev.)
one-time	setup	37.207 s
per app	compile	0.007 s (±0.003 s)
per tx	create	21.639 s (±0.152 s)
	verify	0.027 s (±0.003 s)

(see Fig. 1) closely follow the "user-defined asset" and "intent-based DEX" examples of ZEXE [14]. In contrast to ZEXE, where these apps are implemented as low-level predicates, they are naturally expressed in Zapper's frontend. The other apps are our creations. Being a core component, the Coin class is used across multiple apps.

In all applications, Zapper's data and identity privacy properties are key. For example, "Ticket" allows travelers to punch multi-journey tickets valid for a specific duration after punching, while preventing ticket holders to be traced across journeys.

9.2 Performance

We now evaluate the performance of Zapper.

Parameters For the following experiment, we instantiate the parameters of Zapper as shown in Tab. 3a. Here, N_{fields} is the maximum number of fields per object, excluding `owner`. The Merkle tree height N_{height} is sufficiently large for a real-world deployment and matches the height in Zcash [33]. The other parameters were chosen to support the apps in Tab. 2 with a comfortable margin.

The choice of parameters is subject to a trade-off: larger values enable more complex applications but induce more overhead. In practice, Zapper can support multiple combinations of parameters and prepare a separate proof circuit for each combination, enabling Zapper to select the smallest parameters sufficient to enable any given application. However, it is not obvious how to allow setting N_{height} dynamically, as all applications must operate on the same Merkle tree. To help predict the performance of future applications, §9.3 discusses the effect of individual parameters on performance.

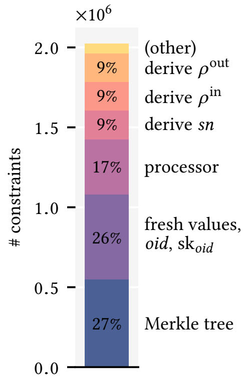
Scenarios For each application in Tab. 2, we create a *scenario* consisting of multiple transactions interacting with the application. For example, for Exchange we first mint two coins and then run `create` and `accept` transactions as visualized in Fig. 1c.

Runtimes Tab. 3b summarizes the performance of Zapper for our scenarios, showing that Zapper transactions are very efficient.

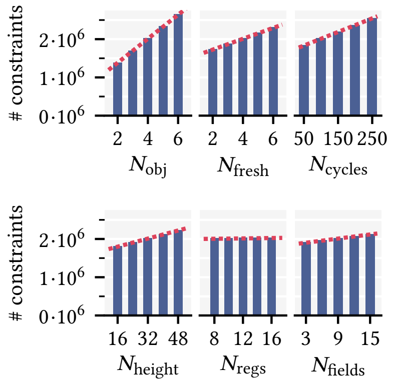
Before executing any transactions, we first initialize a new Zapper ledger ("setup" in Tab. 3b). This global one-time step includes a trusted setup phase for the GM17 [29] zk-SNARKs, which is relatively expensive and dominates the runtime of the step (99.87%).

Next, we compile all applications in Tab. 2 to Zasm code and register this code at the Zapper assembly storage. As shown in Tab. 3b ("compile"), compiling and registering a single application is very efficient as it does not involve any cryptographic operations.

Finally, we execute the scenarios on the Zapper ledger. Executing a transaction consists of two steps. First, the Zapper client locally creates the transaction (§5.2, "create"). The runtime of this step is dominated by zk-SNARK generation (99.97% on avg.). Second, the Zapper executor processes this transaction (§5.3, "verify"), which



(a) Breakdown of component sizes for the parameters in Tab. 3a.



(b) Circuit size when varying individual parameters (🟦) and estimate (⋯) computed by the empirical fit in Eq. (5).

Figure 5: Number of R1CS constraints in the proof circuit ϕ (Alg. 1).

is dominated by zk-SNARK verification (59.1%) and Merkle tree updates (40.6%). While the first step is more expensive, it is only executed by the client. In contrast, the second step is significantly cheaper. This is important, as it will be replicated across many machines when deploying Zapper to a shared ledger.

Runtime Discussion As shown by the low standard deviations in Tab. 3b, the runtimes for creating and verifying transactions are very consistent across all scenarios: because the proof circuit for the zk-SNARK in a transaction is independent of the application, the dominating proof generation times are nearly identical.

Our transaction runtimes are in line with ZEXE [14]. The authors report that transactions require 52.5 s to generate and 0.046 s to verify on a similar machine (3.00 GHz, 24 threads). While we used $N_{\text{obj}} = 4$ and the logic of contracts in Tab. 2, the ZEXE evaluation assumed 2 inputs and 2 outputs, and empty predicates.

Transaction Size Zapper transactions are small, consisting of only 3312 bytes regardless of the called function. This is comparable to ZEXE, whose transactions consist of 968 bytes.

9.3 Proof Circuit Size

The runtimes for zk-SNARK setup and proof generation, which are the dominating parts of "setup" and "create" in Tab. 3b, are linear in the size of the proof circuit. We next analyze this size, measured in the number of rank-1 constraint system (R1CS) constraints.

Size of Individual Components For the parameters in Tab. 3a, the proof circuit (Alg. 1) consists of $2.02\cdot 10^6$ constraints (regardless of the application). In Fig. 5a, we show the sizes of the individual components. The largest two components are the checks of Merkle tree certificates (Lin. 13 in Alg. 1) and preparation of precomputed values (Lin. 2–4 in Alg. 2), as they involve the evaluation of expensive cryptographic hash functions. This is followed by the emulation of processor cycles (Lin. 5–7 in Alg. 2), the derivations of serial numbers *sn*, nonces ρ^{in}, and nonces ρ^{out} (Alg. 1).

Effect of Parameters We next measure the size of the proof circuit for different parameters. This allows gauging the performance of Zapper when parameters are selected dynamically from a set of prepared parameters. In each of the subplots in Fig. 5b, we show the number of constraints when varying a single parameter while setting all other parameters to the values in Tab. 3a. Fig. 5b indicates that N_{obj} has the biggest impact, while N_{regs} and N_{fields} have negligible effects. Overall, the main proof circuit has asymptotic size:

$$O\Big(\underbrace{N_{\text{obj}}(N_{\text{height}} + N_{\text{fields}})}_{\text{Lin. 8–16 \& 18–20 in Alg. 1}} + \underbrace{N_{\text{fresh}}}_{\substack{\text{Lin. 2–4}\\ \text{in Alg. 2}}} + \underbrace{N_{\text{cycles}}(N_{\text{obj}}N_{\text{fields}} + N_{\text{regs}} + N_{\text{fresh}})}_{\text{Lin. 5–8 in Alg. 2}}\Big).$$

To predict the performance of Zapper for given parameters, we have further estimated the constants hidden in the above formulas using an empirical least-square fit. We find that the number of R1CS constraints in the proof circuit can be predicted as:

$$\begin{aligned} &3\,400 + N_{\text{obj}}\left(160\,000 + 3\,300\,N_{\text{height}} + 1\,900\,N_{\text{fields}}\right) + 130\,000\,N_{\text{fresh}} \\ &+ N_{\text{cycles}}\left(1\,600 + 26\,N_{\text{obj}}\,N_{\text{fields}} + 24\,N_{\text{regs}} + 120\,N_{\text{fresh}} + 76\,N_{\text{obj}}\right). \end{aligned} \quad (5)$$

As indicated in Fig. 5b, this prediction is very accurate.

10 LIMITATIONS

We now discuss limitations of Zapper.

First, Zapper only supports Zasm programs respecting its parameters N_{cycles}, N_{obj}, N_{fresh}, N_{fields}, and N_{regs}. If these parameters limit expressivity, developers can increase them (see §9.2).

As discussed in §3.1, Zapper does not natively support control flow, but if-then-else branches can always be rewritten as conditional assignments, and bounded loops can be unrolled. While unbounded loops are not supported, these are already discouraged in non-private smart contracts [19] and can instead be split into individual, bounded-length transactions. Further, Zapper disallows pointer arithmetic and self-modifying code, which are however uncommon in smart contracts. Also, Zasm programs with recursion are disallowed and must be restructured. We expect this is often feasible, e.g. by rewriting tail recursion into loops or by splitting recursive calls into individual non-recursive transactions.

A fundamental limitation of Zapper is that it only allows users to create transactions for which the data of all accessed objects is known. In particular, contracts cannot privately communicate "amongst each other" while keeping the communicated data hidden from all users. To enable this, Zapper would need to leverage additional cryptographic primitives such as homomorphic encryption.

Finally, some applications such as private machine learning are not suitable for Zapper, but can be realized using secure multi-party computation (MPC) or fully homomorphic encryption (FHE). However, these techniques are generally less scalable in terms of the number of involved parties, communication, or computation.

11 COMPARISON TO ZEXE

We now elaborate on the shortcomings of ZEXE [14] compared to Zapper (see also §1). ZEXE specifies smart contracts using records and predicates. It provides strong data, identity, and function privacy based on nested zk-SNARKs and ideas from Zerocash [47].

Application Vulnerabilities We have discovered two vulnerabilities in ZEXE's motivating example of a DEX [14, §I-A], which allow an attacker to lock a coin belonging to another user. Both attacks have been confirmed by the authors of ZEXE ([15, Acknowledgments] and private correspondence). Note that our notion of availability (§7) prevents such attacks by design.

First, as ZEXE only stores commitments of data, transferring data to another user requires out-of-band communication. As this can be denied by a malicious user, DEX is subject to a "denial-of-funds" attack, where the attacker accepts an offer but refuses to share the information required to receive the attacker's coin. To prevent this attack, the ZEXE authors recommended adapting DEX to store the encrypted output record in a public memorandum field and extending the predicates to check for correct encryption [15, Remark 6.1]. However, only developers intimately familiar with key-private and NIZK-friendly encryption can implement this securely and efficiently. Even if cryptographic experts provide according cryptographic primitives, developers still need to decide whether to use these, depending on the application (note that the attack does not exist for the coin itself, but only when the coin is used in a DEX). In contrast, Zapper by design uses an appropriate encryption scheme instead of commitments (see §8).

Second, we have identified a "lock-out" attack on the DEX application. Like Zapper, ZEXE pads input records by dead records. However, unlike Zapper, ZEXE does not enforce that their serial nonces ρ^{in} are globally unique (see Lin. 15 in Alg. 1). Thus, an attacker knowing the shared address secret key of a DEX record can block access to the record by consuming a dead record with a conflicting serial nonce and serial number, thus blocking the coin to be traded by the DEX indefinitely. We expect that our attack can be prevented in ZEXE by constraining serial nonces of dead records.

Lack of Modularity ZEXE obstructs modular development, as cooperating applications must typically be mutually aware of each other to ensure that their logic cannot be violated in the future.

For example, ZEXE's motivating example of a DEX [14, §I-A] introduces a tight coupling between DexOffer and Coin. Specifically, to prevent adversaries from creating coins out of thin air, their *birth predicate* (which must be satisfied when creating a new coin) ensures that coins can only be created in exchange for existing coins (identified by their birth predicate). However, because a DexOffer record cannot "own" a coin record (ZEXE has no concept of ownership), a newly created DexOffer consumes the coin to be traded. In turn, accepting a DexOffer hence re-creates the previously consumed coin, which requires the DexOffer to have the same birth predicate as coin (see above), thus essentially merging both applications. Note that adapting the Coin birth predicate to allow consuming non-merged DexOffer objects would require trusting that DexOffer does not create coins out of thin air.

Following the above pattern, all potential applications using coins must be anticipated and implemented in advance—a severe limitation in practice.

In contrast, Zapper's object ownership feature and access control policies allow classes to be developed independently and modularly.

Trusted Setup and Usability ZEXE requires a separate trusted setup for *each* application, which when performed by dishonest parties allows violating correctness. [4] In contrast, Zapper only requires a single trusted setup for its application-agnostic proof circuit ϕ. Also, ZEXE relies on a non-standard programming model in terms of *predicates*, while the programming model of Zapper is closer to the most widely used smart contract language Ethereum.

[4]This problem can be partially mitigated by an expensive multi-party computation (MPC), assuming trustworthy participants can be found for every new application.

Function Privacy Unlike Zapper, ZEXE hides the function being executed in a transaction. However, this is often not required in practice (see Tab. 2). Still, future work could extend Zapper to function privacy by providing Zasm instructions as private inputs to the proof circuit and performing class registrations in zero-knowledge.

12 RELATED WORK

Private Cryptocurrencies A long line of work proposes anonymous payment systems, where the participants involved in a transaction are hidden [16, 21, 25, 30, 33, 42, 44, 47]. In contrast to Zapper, these systems focus on payments only and do not support general smart contracts. However, Zapper does rely on the techniques of Zerocash [33, 47] to hide the objects accessed in a transaction.

Private Smart Contracts Various works bring privacy to smart contracts. Hawk [37], Arbitrum [35], Ekiden [17], and FastKitten [20] assume a strong trust model by relying on trusted managers or hardware. In contrast, Zapper only relies on a single trusted zk-SNARK setup. While zkHawk [3] and V-zkHawk [4] weaken the trust assumption of Hawk, they require interactive parties. SmartFHE [48], zkay [52], and ZeeStar [51] provide data privacy for smart contracts with weak trust assumptions, but do not target identity privacy and leak the accessed memory locations. We have already discussed ZEXE [14] separately in §11.

Zero-knowledge Rollups Complementary to Zapper, *ZK rollups* such as StarkNet [50], zkSync [57], and Aztec [53] combine multiple smart contract transactions into a single one using NIZK proofs. However, to date, StarkNet and zkSync do not provide privacy [49, 56]. While the announced Aztec "ZK-ZK-Rollup" system [54] aims to achieve *private* rollups, it has not yet been released.

Zero-knowledge Processors The idea of executing a processor in zero-knowledge has been thoroughly studied before Zapper. For instance, BubbleRAM [31], BubbleCache [32], and ZKarray [27] present zero-knowledge processors with efficient RAM. However, unlike Zapper, they target an interactive setting.

Similar to Zapper, the TinyRAM line of work [7–9] emulates a processor inside (non-interactive) zk-SNARKs. As discussed in §6, the Zapper processor applies techniques introduced by these works. Zapper could potentially be extended, e.g., to target a von Neumann architecture, allowing powerful techniques such as self-modifying code [9]. However, as we demonstrate in §9, the Zapper processor already supports realistic applications.

13 CONCLUSION

We have presented Zapper, a privacy-focused smart contract system that allows developers to express smart contracts in an intuitive frontend. Zapper is highly efficient and achieves data and identity privacy, correctness, access control, integrity, and availability.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive feedback and Pratyush Mishra for helpful discussions on ZEXE's vulnerabilities.

REFERENCES

[1] Arkworks Contributors. 2022. *arkworks zkSNARK ecosystem.* https://arkworks.rs

[2] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. 2013. BLAKE2: Simpler, Smaller, Fast as MD5. In *Applied Cryptography and Network Security*, Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 119–135.

[3] Aritra Banerjee, Michael Clear, and Hitesh Tewari. 2021. zkHawk: Practical Private Smart Contracts from MPC-based Hawk. In *2021 3rd Conference on Blockchain Research Applications for Innovative Networks and Services (BRAINS).* 245–248. https://doi.org/10.1109/BRAINS52497.2021.9569822

[4] Aritra Banerjee and Hitesh Tewari. 2022. Multiverse of HawkNess: A Universally-Composable MPC-based Hawk Variant. Cryptology ePrint Archive, Report 2022/421. https://ia.cr/2022/421.

[5] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. 2002. Constructing Elliptic Curves with Prescribed Embedding Degrees. Cryptology ePrint Archive, Report 2002/088. https://ia.cr/2002/088.

[6] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. 2001. Key-Privacy in Public-Key Encryption. In *Advances in Cryptology — ASIACRYPT 2001*, Colin Boyd (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 566–582.

[7] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *Advances in Cryptology – CRYPTO 2013*, Ran Canetti and Juan A. Garay (Eds.). Vol. 8043. Springer Berlin Heidelberg, Berlin, Heidelberg, 90–108. https://doi.org/10.1007/978-3-642-40084-1_6

[8] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Scalable Zero Knowledge via Cycles of Elliptic Curves. Cryptology ePrint Archive, Report 2014/595. https://ia.cr/2014/595.

[9] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *23rd USENIX Security Symposium (USENIX Security 14).* USENIX Association, San Diego, CA, 781–796. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2012. Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In *Selected Areas in Cryptography*, Ali Miri and Serge Vaudenay (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 320–337.

[11] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2012. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference on - ITCS '12.* ACM Press, Cambridge, Massachusetts, 326–349. https://doi.org/10.1145/2090236.2090263

[12] Manuel Blum, Paul Feldman, and Silvio Micali. 1988. Non-interactive zero-knowledge and its applications. In *Proceedings of the twentieth annual ACM symposium on Theory of computing - STOC '88.* ACM Press, Chicago, Illinois, United States, 103–112. https://doi.org/10.1145/62212.62222

[13] Sean Bowe. 2017. *Jujub elliptic curve.* https://github.com/zkcrypto/jubjub

[14] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. 2020. ZEXE: Enabling Decentralized Private Computation. In *2020 IEEE Symposium on Security and Privacy (SP).* IEEE Computer Society, Los Alamitos, CA, USA, 1114–1131. https://doi.org/10.1109/SP40000.2020.00050 ISSN: 2375-1207.

[15] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2021. ZEXE: Enabling Decentralized Private Computation. Cryptology ePrint Archive, Report 2018/962 (updated 2021-03-30). https://ia.cr/2018/962.

[16] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. 2020. Zether: Towards Privacy in a Smart Contract World. In *Financial Cryptography and Data Security*, Joseph Bonneau and Nadia Heninger (Eds.). Springer International Publishing, Cham, 423–443.

[17] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2018. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *CoRR* abs/1804.05141 (2018). http://arxiv.org/abs/1804.05141

[18] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Advances in Cryptology - EUROCRYPT 2020*, Anne Canteaut and Yuval Ishai (Eds.). Springer International Publishing, Cham, 738–768.

[19] ConsenSys. 2022. Ethereum Smart Contract Best Practices: Denial of Service. https://consensys.github.io/smart-contract-best-practices/attacks/denial-of-service/ accessed: 2022-07-28.

[20] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. 2019. FastKitten: Practical Smart Contracts on Bitcoin. In *28th USENIX Security Symposium (USENIX Security 19).* USENIX Association, Santa Clara, CA, 801–818. https://www.usenix.org/conference/usenixsecurity19/presentation/das

[21] Benjamin E. Diamond. 2021. Many-out-of-Many Proofs and Applications to Anonymous Zether. In *2021 IEEE Symposium on Security and Privacy (SP).* 1800–1817. https://doi.org/10.1109/SP40001.2021.00026

[22] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *13th USENIX Security Symposium (USENIX Security 04).* USENIX Association, San Diego, CA. https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router

[23] T. Elgamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory* 31, 4 (1985), 469–472. https://doi.org/10.1109/TIT.1985.1057074

[24] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. 2020. SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain. In *Financial Cryptography and Data Security*. Springer International Publishing, Cham, 170–189.

[25] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. 2019. Quisquis: A New Design for Anonymous Cryptocurrencies. In *Advances in Cryptology – ASIACRYPT 2019*, Steven D. Galbraith and Shiho Moriai (Eds.). Springer International Publishing, Cham, 649–678. https://link.springer.com/chapter/10.1007/978-3-030-34578-5_23

[26] Uriel Feige, Dror Lapidot, and Adi Shamir. 1999. Multiple Non-Interactive Zero Knowledge Proofs Under General Assumptions. *SIAM J. Comput.* 29, 1 (Sept. 1999), 1–28. https://doi.org/10.1137/S0097539792230010

[27] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. 2021. Constant-Overhead Zero-Knowledge for RAM Programs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security.* ACM, Virtual Event Republic of Korea, 178–191. https://doi.org/10.1145/3460120.3484800

[28] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium (USENIX Security 21).* USENIX Association, 519–535. https://www.usenix.org/conference/usenixsecurity21/presentation/grassi

[29] Jens Groth and Mary Maller. 2017. Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs. Cryptology ePrint Archive, Report 2017/540. https://ia.cr/2017/540.

[30] Zhangshuang Guan, Zhiguo Wan, Yang Yang, Yan Zhou, and Butian Huang. 2020. BlockMaze: An Efficient Privacy-Preserving Account-Model Blockchain Based on zk-SNARKs. *IEEE Transactions on Dependable and Secure Computing* (2020). https://doi.org/10.1109/TDSC.2020.3025129

[31] David Heath and Vladimir Kolesnikov. 2020. A 2.1 KHz Zero-Knowledge Processor with BubbleRAM. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security.* ACM, Virtual Event USA, 2055–2074. https://doi.org/10.1145/3372297.3417283

[32] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. 2021. Zero Knowledge for Everything and Everyone: Fast ZK Processor with Cached ORAM for ANSI C Programs. In *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, San Francisco, CA, USA, 1538–1556. https://doi.org/10.1109/SP40001.2021.00089

[33] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. 2021. *Zcash Protocol Specification.* Technical Report. Electric Coin Company. https://raw.githubusercontent.com/zcash/zips/master/protocol/protocol.pdf Accessed: 2022-04-28.

[34] Dave Jevans. 2020. CipherTrace Files Two Monero Cryptocurrency Tracing Patents. https://ciphertrace.com/ciphertrace-files-two-monero-cryptocurrency-tracing-patents/. Accessed: 2022-04-15.

[35] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. 2018. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18).* USENIX Association, Baltimore, MD, 1353–1370. https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner

[36] Dmitry Khovratovich. 2019. Encryption with Poseidion. https://dusk.network/uploads/Encryption-with-Poseidon.pdf

[37] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *2016 IEEE Symposium on Security and Privacy.* IEEE, 839–858. https://doi.org/10.1109/SP.2016.55

[38] B. Laurie, A. Langley, and E. Kasper. 2013. *Certificate Transparency.* RFC 6962.

[39] Toghrul Maharramov, Dmitry Khovratovich, and Emanuele Francioni. 2021. The Dusk Network Whitepaper. https://dusk.network/uploads/The_Dusk_Network_Whitepaper_v3_0_0.pdf Accessed: 2022-04-28.

[40] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO '87).* Springer-Verlag, Berlin, Heidelberg, 369–378.

[41] S. Micali, M. Rabin, and J. Kilian. 2003. Zero-knowledge sets. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE Computer. Soc, Cambridge, MA, USA, 80–91. https://doi.org/10.1109/SFCS.2003.1238183

[42] I. Miers, C. Garman, M. Green, and A. D. Rubin. 2013. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy.* IEEE, Berkeley, CA, 397–411. https://doi.org/10.1109/SP.2013.34

[43] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, and Nicolas Christin. 2018. An Empirical Analysis of Traceability in the Monero

Blockchain. *arXiv:1704.04299 [cs]* (April 2018). http://arxiv.org/abs/1704.04299 arXiv: 1704.04299.

[44] Shen Noether. 2015. Ring Signature Confidential Transactions for Monero. Cryptology ePrint Archive, Report 2015/1098. https://eprint.iacr.org/2015/1098.

[45] Andrew Norry. 2020. The History of the Mt Gox Hack: Bitcoin's Biggest Heist. Blockonomi. https://blockonomi.com/mt-gox-hack/ accessed: 2022-04-20.

[46] Fergal Reid and Martin Harrigan. 2012. An Analysis of Anonymity in the Bitcoin System. *arXiv:1107.4524 [physics]* (May 2012). http://arxiv.org/abs/1107.4524 arXiv: 1107.4524.

[47] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. 459–474.

[48] Ravital Solomon and Ghada Almashaqbeh. 2021. smartFHE: Privacy-Preserving Smart Contracts from Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2021/133. https://eprint.iacr.org/2021/133.

[49] StarkNet. 2022. Privacy on StarkNet. https://starknet.io/faq/privacy-on-starknet/ accessed: 2022-05-01.

[50] StarkNet. 2022. StarkNet Main Webpage. https://starknet.io/ accessed: 2022-05-01.

[51] Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. 2022. ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zero-knowledge Proofs. In *2022 IEEE Symposium on Security and Privacy (SP)*.

[52] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. 2019. zkay: Specifying and Enforcing Data Privacy in Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, London United Kingdom, 1759–1776. https://doi.org/10.1145/3319535.3363222

[53] Zachary Williamson. 2021. The AZTEC Protocol. https://github.com/AztecProtocol/AZTEC/blob/master/AZTEC.pdf accessed: 2022-05-01.

[54] Zac Williamson. 2021. Aztec's ZK-ZK-Rollup, looking behind the cryptocurtain. Medium. https://medium.com/aztec-protocol/aztecs-zk-zk-rollup-looking-behind-the-cryptocurtain-2b8af1fca619 accessed: 2022-05-01.

[55] Wolfie Zhao. 2018. Bithumb $31 Million Crypto Exchange Hack: What We Know (And Don't). CoinDesk. https://www.coindesk.com/markets/2018/06/20/bithumb-31-million-crypto-exchange-hack-what-we-know-and-dont/ accessed: 2022-04-20.

[56] zkSync. 2022. zkSync Documentation: Privacy. https://docs.zksync.io/userdocs/privacy.html accessed: 2022-05-01.

[57] zkSync. 2022. zkSync Main Webpage. https://zksync.io/ accessed: 2022-05-01.