Zapper: Smart Contracts with Data and Identity Privacy

Samuel Steffen  Benjamin Bichsel  Martin Vechev

ETH Zurich, Switzerland
Introduction: Privacy for Smart Contracts

**Permissionless Blockchain**

**Goal:** hide data and user identities

**Existing work**
- weak privacy guarantees
- strong trust assumptions
- manual instantiation of crypto
Idea

Zerocash [Sasson et al., 2014] / Zcash

- Strong privacy guarantees
- Not programmable
- Make programmable

But avoid limitations of previous work (e.g., ZEXE [Bowe et al., 2020])
call: Coin.transfer()
receiver: obj
args: 4, 2
sender: Alice
call: `Coin.transfer()`
receiver: `obj`
args: `4, 2`
sender: `Alice`
Example: Coin Puzzle

class Coin(Contract):
    val: Uint
    # owner: Address

def transfer(self, to: Address):
    require(self.owner == self.me)
    self.owner = to
Example: Coin Puzzle

class Coin(Contract):
    val: Uint
    # owner: Address

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

Alice → Bob
**Example: Coin Puzzle**

```python
class Coin(Contract):
    val: Uint
    # owner: Address

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

Alice

Bob

```python
class Wrapper(Contract):
    coin: Coin
    # owner: Address

def constructor(self, coin: Coin, owner: Address):
    self.owner = owner
    self.coin = coin
    coin.transfer(self.address)

def puzzle(self, sol: Uint) -> Bool:
    ...

def open(self, sol: Uint):
    require(self.puzzle(sol))
    self.coin.transfer(self.me, sender_is_self=True)
```
class Coin(Contract):
    val: Uint
    # owner: Address

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

Example: Coin Puzzle

class Wrapper(Contract):
    coin: Coin
    # owner: Address

    def constructor(self, coin: Coin, owner: Address):
        self.owner = owner
        self.coin = coin
        coin.transfer(self.address)

    def puzzle(self, sol: Uint) -> Bool:
        ...

    def open(self, sol: Uint):
        require(self.puzzle(sol))
        self.coin.transfer(self.me, sender_is_self=True)
class Coin(Contract):
    val: Uint
    # owner: Address

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

Example: Coin Puzzle

class Wrapper(Contract):
    coin: Coin
    # owner: Address

    def constructor(self, coin: Coin, owner: Address):
        self.owner = owner
        self.coin = coin
        coin.transfer(self.address)

    def puzzle(self, sol: Uint) -> Bool:
        ...

    def open(self, sol: Uint):
        require(self.puzzle(sol))
        self.coin.transfer(self.me, sender_is_self=True)
Example: Coin Puzzle

```python
class Coin(Contract):
    val: Uint
    # owner: Address

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

Example: Coin Puzzle

class Wrapper(Contract):
    coin: Coin
    # owner: Address

    def constructor(self, coin: Coin, owner: Address):
        self.owner = owner
        self.coin = coin
        coin.transfer(self.address)

    def puzzle(self, sol: Uint) -> Bool:
        ...

    def open(self, sol: Uint):
        require(self.puzzle(sol))
        self.coin.transfer(self.me, sender_is_self=True)
```

Alice  Bob
Example: Coin Puzzle

class Coin(Contract):
    val: Uint
    # owner: Address

def transfer(self, to: Address):
    require(self.owner == self.me)
    self.owner = to
def foo(self, to: Address):
    require(self.owner == self.me)
    self.other.bar()
    self.owner = to

static checks, insert necessary runtime checks

interaction between classes only via function calls

type check

access control

inline calls

Distributed Ledger

CID tmp0 self
EQ tmp1 tmp0 'Foo'
REQ tmp1
LOAD tmp2 self 'owner'
EQ tmp3 tmp2 me
REQ tmp3
LOAD tmp4 self 'other'

STORE arg0 self 'owner'

no control-flow (but CMOV)
no loops
Storing and Updating Objects

Merkle tree

**I correctly...**
- *accessed* the required object states
- *invalidated* these states
- *updated* the states according to the called function
- *encrypted* the new states

**generalize Zerocash / Zcash to objects**

**avoid pitfalls** (very technical)

**Coin#2022.transfer(Bob)**

**zk-SNARK**

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Coin&quot; #2022 owner: Alice val: 1</td>
<td>&quot;Coin&quot; #2022 owner: Bob val: 1</td>
</tr>
</tbody>
</table>
Zero-knowledge Processor

assembly program ...

public inputs

secret inputs

arguments

input object states

output object states ...

verified

precomputation and prefetching objects

same zk-SNARK circuit for *all* programs *
Security Properties

- **Data and Identity Privacy**: Object accesses, data, sender, and arguments are hidden.

- **Correctness**: Cannot violate class logic.

- **Integrity**: Cannot tamper with or replay transactions.

- **Availability**: Cannot block valid transactions.

Simulation-based indistinguishability proof identified and fixed 2 attacks on ZEXE.
Evaluation

Available on eth-sri/zapper on idealized ledger

Expressiveness

- Coin
- Decentralized Exchange
- Private Auction
- Double-blind Peer-review
- ...

Efficiency

On commodity desktop

- < 0.01 s compilation
- ≈ 22 s tx generation
- < 0.03 s tx verification
- 99.9 % proof generation
- w/o consensus

Available on eth-sri/zapper on idealized ledger
class Coin(Contract):
    val: Uint
    # owner: Address

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

    call:
    receiver: Coin.transfer()
    args:
        obj: 4, 2
        sender: Alice