ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zero-knowledge Proofs

Samuel Steffen    Benjamin Bichsel    Roger Baumgartner    Martin Vechev

ETH Zurich, Switzerland

{samuel.steffen, benjamin.bichsel, martin.vechev}@inf.ethz.ch, rogerb@student.ethz.ch
Motivation

everyone can observe all data

smart contract  stored data

public blockchain
Data Privacy

smart contract

public blockchain

stored data

data hidden
Existing Works for Data Privacy

“30 seconds version”

Private cryptocurrencies, Zether [FC 20], ...

no general smart contracts
Existing Works for Data Privacy

Private cryptocurrencies, Zether [FC 20], ...

Hawk [S&P 16], Arbitrum [Usenix 18], Ekiden [Euro S&P 19], ...

“30 seconds version”

no general smart contracts

trusted managers or hardware
Existing Works for Data Privacy

- Private cryptocurrencies, Zether [FC 20], ...
  - no general smart contracts

- Hawk [S&P 16], Arbitrum [Usenix 18], Ekiden [Euro S&P 19], ...
  - trusted managers or hardware

- ZEXE [S&P 20], smartFHE [ePrint 21]
  - cryptographic expertise required

“30 seconds version”
Existing Works for Data Privacy

- Private cryptocurrencies, Zether [FC 20], ...
  - no general smart contracts

- Hawk [S&P 16], Arbitrum [Usenix 18], Ekiden [Euro S&P 19], ...
  - trusted managers or hardware

- ZEXE [S&P 20], smartFHE [ePrint 21]
  - cryptographic expertise required

- zkay [CCS 19]
  - limited expressivity

“30 seconds version”
This Work

ZeeStar

- Smart contracts
- Weak trust assumptions
- No cryptographic expertise required
- High expressivity
- On Ethereum
This Work

ZeeStar

*conceptually*: extends zkay by homomorphic encryption

- smart contracts
- weak trust assumptions
- no cryptographic expertise required
- high expressivity
- on Ethereum
Updating Encrypted Balances

Alice’s balance
42

Bob’s balance
56
Updating Encrypted Balances

Alice’s balance

Bob’s balance
Updating Encrypted Balances

Alice’s balance

\[ \text{Alice’s balance: df92...0e} \]

\[ \text{pk}_{\text{Alice}} \]

Bob’s balance

\[ \text{Bob’s balance: 130a...14} \]

\[ \text{pk}_{\text{Bob}} \]

Alice: “-1”

Alice: “+1”

use NIZK proofs and homomorphic encryption
Updating Encrypted Balances

self-owned

plaintext operation

df92...0e

42

- 1 = 41

decrypt

encrypt

7901...a6
Updating Encrypted Balances

plaintext operation

self-owned

“state update is correct”

NIZK proof
Updating Encrypted Balances

self-owned

foreign

plaintext operation

NIZK proof

df92...0e → 42 - 1 = 41 → 7901...a6

homomorphic operation

130a...14 ⊕ 9c8f...55 = 0aa1...bb
Updating Encrypted Balances

self-owned

<table>
<thead>
<tr>
<th>df92...0e</th>
<th>→</th>
<th>42</th>
<th>→</th>
<th>-1</th>
<th>=</th>
<th>41</th>
<th>→</th>
<th>7901...a6</th>
</tr>
</thead>
<tbody>
<tr>
<td>130a...14</td>
<td>⊕</td>
<td>9c8f...55</td>
<td>=</td>
<td>0aa1...bb</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

foreign

homomorphic operation

NIZK proof
Goal: Automate

How to automate?

NIZK proof

Homomorphic operation

Self-owned

Foreign

plaintext operation
Goal: Automate

plaintext operation

how to automate?

NIZK proof

self-owned

how to distinguish?

foreign

interactions?

homomorphic operation

130a...14

1

9c8f...55

= 0aa1...bb

df92...0e

42 - 1 = 41

7901...a6
Goal: Automate

self-owned
how to distinguish?
restrictions?
interactions?

foreign
homomorphic operation

how to automate?

NIZK proof

plaintext operation

df92...0e
42 - 1 = 41
7901...a6

130a...14 ⊕ 9c8f...55 = 0aa1...bb
Goal: Automate

- plaintext operation

- how to automate?

- how to distinguish?

- restrictions?

- interactions?

- homomorphic operation

130a...14 \oplus 9c8f...55 = 0aa1...bb

NIZK proof

what to prove?

efficiency?
Overview: ZeeStar

ZeeStar contract

logic and privacy annotations
Overview: ZeeStar

- ZeeStar contract
- Logic and privacy annotations
- ZeeStar compiler
- Additively homomorphic encryption
- zk-SNARK
- Solidity contract
**Privacy Types**

- **self-owned**
  - how to distinguish?
  - restrictions?
  - how to automate?

- **foreign**
  - interactions?
  - homomorphic operation
  - what to prove?
  - efficiency?
  - NIZK proof

plaintext operation

Self-owned privacy types include:
- how to distinguish?
- restrictions?
- how to automate?

Foreign privacy types include:
- interactions?
- homomorphic operation
- what to prove?
- efficiency?
- NIZK proof

Examples:
- df92...0e
- 42 - 1 = 41
- 7901...a6

Operations:
- 130a...14 + 9c8f...55 = 0aa1...bb
Example: Private Balances

```solidity
contract Balances {
    mapping(address => uint) bal;

    function transfer(uint val, address to) {
        require(val <= bal[me]);
        bal[me] = bal[me] - val;
        bal[to] = bal[to] + val;
    }
}
```
Privacy Types

only party allowed to see data

datatype@owner

from zkay
Privacy Annotations and Types

```solidity
contract Balances {
    mapping(address => uint) bal;

    function transfer(uint val, address to) {
        require(reveal(val <= bal[me], all));
        bal[me] = bal[me] - val;
        bal[to] = bal[to] + reveal(val, to);
    }
}
```

from zkay, but adapted ZeeStar
Privacy Annotations and Types

contract Balances {
    mapping(address => uint) bal;

    function transfer(uint me_val, address to) {
        require(reveal(me_val <= bal[me]), all));
        bal[me] = bal[me] - me_val;
        bal[to] = bal[to] + reveal(me_val, to);
    }
}
contract Balances {
    mapping(address => uint) bal;

    function transfer(uint@me val, address to) {
        require(reveal(val <= bal[me], all));
        bal[me] = bal[me] - val;
        bal[to] = bal[to] + reveal(val, to);
    }
}

from zkay, but adapted
Privacy Annotations and Types

```solidity
contract Balances {
    mapping(address!x => uint@x) bal;

    function transfer(uint@me val, address to) {
        require(reveal(val <= bal[me], all));
        bal[me] = bal[me] - val;
        bal[to] = bal[to] + reveal(val, to);
    }
}
```

from zkay, but adapted

modify self-owned value
Privacy Annotations and Types

contract Balances {
    mapping(address!x => uint@x) bal;

    function transfer(uint@me val, address to) {
        require(reveal(val <= bal[me], all));
        bal[me] = bal[me] - val;
        bal[to] = bal[to] + reveal(val, to);
    }
}

from zkay, but adapted

modify foreign value

change privacy type
@me → @to
Privacy Annotations and Types

contract Balances {
    mapping(address => uint) bal;

    function transfer(uint @meval, address to) {
        require(reveal(val <= bal[me], all));
        bal[me] = bal[me] - @meval;
        bal[to] = bal[to] + reveal(val, to);
    }
}

from zkay, but adapted

modify foreign value

allowed (disallowed in zkay)

change privacy type @me → @to
Privacy Annotations and Types

```solidity
contract Balances {
    mapping(address => uint) bal;

    function transfer(uint value, address to) {
        require(reveal(value <= bal[me], all));
        bal[me] = bal[me] - value;
        bal[to] = bal[to] + reveal(value, to);
    }
}
```

from zkay, but adapted

modify foreign value

allowed (disallowed in zkay)

change privacy type @me → @to

type error (cannot realize)

bal[bob] + bal[charlie]
contract Balances {
    mapping(address => uint) bal;

    function transfer(uint me val, address to) {
        require(reveal(val <= bal[me], all));
        bal[me] = bal[me] - val;
        bal[to] = bal[to] + reveal(val, to);
    }
}
Privacy Annotations and Types

```solidity
contract Balances {
    mapping(address => uint) bal;

    function transfer(uint val, address to) {
        require(reveal(val <= bal[me], all));
        bal[me] = bal[me] - val;
        bal[to] = bal[to] + reveal(val, to);
    }
}
```

from zkay, but adapted

no cryptographic expertise required
Compilation

how to distinguish?

restrictions?

homomorphic operation

130a...14 ⊕ 9c8f...55 = 0aa1...bb

how to automate?

what to prove?

NIZK proof

efficiency?
function transfer(uint@me val, address to) {
    require(reveal(val <= bal[me], all));
    bal[me] = bal[me] - val;
    bal[to] = bal[to] + reveal(val, to);
}

function transfer(...) {
    require(ok);
    bal[me] = new_me;
    bal[to] = new_to;
    verify(proof, ...);
}
Compilation

function transfer(uint@me val, address to) {
    require(reveal(val <= bal[me], all));
    bal[me] = bal[me] - val;
    bal[to] = bal[to] + reveal(val, to);
}

function transfer(...) {
    require(ok);
    bal[me] = new_me;
    bal[to] = new_to;
    verify(proof, ...);
}

ZeeStar

owned by sender (@me)

encrypted for owner (@me)

Solidity
Compilation

```
function transfer(uint @me val, address to) {
    require(reveal(val, me) <= bal[me], all));
    bal[me] = bal[me] - val;
    bal[to] = bal[to] + reveal(val, to);
}
```

```
function transfer(...) {
    require(ok);
    bal[me] = new_me;
    bal[to] = new_to;
    verify(proof, ...);
}
```
Compilation

```solidity
function transfer(uint@me val, address to) {
    require(reveal(val <= bal[me], all));
    bal[me] = bal[me] - val;
    bal[to] = bal[to] + reveal(val, to);
}
```

to prove:

```solidity
val' ← Dec(val, sk_me)
new_me == Enc(Dec(bal[me], sk_me) - val', pk_me, r1)
```
Compilation

```solidity
function transfer(uint@me val, address to) {
    require(reveal(val <= bal[me], all));
    bal[me] = bal[me] - val;
    bal[to] = bal[to] + reveal(val, to);
}
```

to prove:

```
val’ ← Dec(val, sk_me)
new_me == Enc(Dec(bal[me], sk_me) - val’, pk_me, r1)
new_to == bal[to] ⊕ Enc(val’, pk_to, r2)
```

dom heteromorphic addition *inside* proof (to reduce gas costs)
Compilation

![Compilation Diagram]

```solidity
function transfer(uint256 val, address to) {
    require(reveal(val <= bal[me], all));
    bal[me] = bal[me] - val;
    bal[to] = bal[to] + reveal(val, to);
}

function transfer(...) {
    require(ok);
    bal[me] = new_me;
    bal[to] = new_to;
    verify(proof, ...);
}

\[ val' \leftarrow \text{Dec}(val, sk_{me}) \]
\[ \text{ok} = (val' \leq \text{Dec}(bal[me], sk_{me})) \]
\[ \text{new}_\text{me} = \text{Enc}(\text{Dec}(bal[me], sk_{me}) - val', pk_{me}, r1) \]
\[ \text{new}_\text{to} = bal[to] \oplus \text{Enc}(val', pk_{to}, r2) \]
Compilation

function transfer(uint@me val, address to) {
  require(reveal(val <= bal[me], all));
  bal[me] = bal[me] - val;
  bal[to] = bal[to] + reveal(val, to);
}

function transfer(…) {
  require(ok);
  bal[me] = new_me;
  bal[to] = new_to;
  verify(proof, …);
}

to prove:

\[
\begin{align*}
\text{val' } & \leftarrow \text{Dec(val, sk}_\text{me} ) \\
\text{ok } & = (\text{val'} \leq \text{Dec(bal[me], sk}_\text{me} )) \\
\text{new}_\text{me} & = \text{Enc(Dec(bal[me], sk}_\text{me} ) - \text{val'}, pk}_\text{me}, r1) \\
\text{new}_\text{to} & = \text{bal[to]} \oplus \text{Enc(val', pk}_\text{to}, r2) \\
\end{align*}
\]
Compilation

```solidity
function transfer(uint@me val, address to) {
    require(reveal(val <= bal[me], all));
    bal[me] = bal[me] - val;
    bal[to] = bal[to] * reveal(val, to);
}
```

also: multiplication by self-owned or public value

```solidity
function transfer(...) {
    require(ok);
    bal[me] = new_me;
    bal[to] = new_to;
    verify(proof, ...);
}
```
Algorithm 1 Transforming Function Bodies

1: procedure TRANSFORM(f)
2:   \( \mathcal{C}_f = \{ \} \)
3:   for each require\( (e) \) or id = \( e \) in the body of \( f \) do
4:     TRANSFORMEXPR\( (e, f, \mathcal{C}_f) \)
5:   return \( \mathcal{C}_f \)
6:
7: procedure TRANSFORMEXPR\( (e, f, \mathcal{C}_f) \)
8:   if \( e \) has privacy type \( \alpha \neq \text{all} \) then
9:     add new function argument \( \arg \) to \( f \)
10:    replace \( e \) by variable \( \arg \)
11:    add \( \arg \equiv \alpha \) to \( \mathcal{C}_f \)
12: else (\( e \) is public)
13:   for each node \( e_i \) visited during BFS over \( e \) do
14:     if \( e_i \) has the form \( \text{reveal}(\epsilon, \text{all}) \) then
15:         add new function argument \( \arg_i \) to \( f \)
16:        replace subtree rooted at \( e_i \) by variable \( \arg_i \)
17:        add \( \arg_i \equiv \text{all} \) \( e' \) to \( \mathcal{C}_f \)

\[
T_{\text{plan}}(e) = e
\]
(11)
\[
T_{\text{plan}}(\text{me}) = \text{me}
\]
(12)
\[
T_{\text{plan}}(e_1 \text{ op } e_2) = T_{\text{plan}}(e_1) \text{ op } T_{\text{plan}}(e_2)
\]
(13)
\[
T_{\text{plan}}(\text{reveal}(e, \alpha)) = T_{\text{plan}}(e)
\]
(14)
\[
T_{\text{plan}}(\text{id}) = \begin{cases} \text{id}, & \text{if id owned by } \alpha \\ \bot, & \text{otherwise} \end{cases}
\]
(15)

Fig. 6: Transforming constraint directives to constraints.

\[
T_\alpha(e) = \text{Enc}_{\mathcal{C}_\alpha}(e)
\]
(16)
\[
T_\alpha(\text{me}) = \text{Enc}_{\mathcal{C}_\alpha}(\text{me})
\]
(17)
\[
T_\alpha(e) = \begin{cases} \text{Enc}_{\mathcal{C}_\alpha}(T_{\text{plan}}(e)) & \text{if } e \text{ public} \\ \bot, & \text{otherwise} \end{cases}
\]
(18)
\[
T_\alpha(e_1 \text{ op } e_2) = \begin{cases} T_\alpha(e_1) \oplus T_\alpha(e_2) & \text{if } \text{op} = + \\ T_\alpha(e_1) \ominus T_\alpha(e_2) & \text{if } \text{op} = - \\ \bot, & \text{otherwise} \end{cases}
\]
(19)
\[
T_\alpha(\text{reveal}(e, \alpha')) = \begin{cases} \text{Enc}_{\mathcal{C}_\alpha}(T_{\text{plan}}(e)) & \text{if } \alpha = \alpha' \\ \bot, & \text{otherwise} \end{cases}
\]
(20)

Fig. 7: Recursive expression transformation using \( T_{\text{plan}} \). Undefined cases (\( \bot \)) never apply for well-typed contracts.

\[
T_\alpha(e_1 * e_2) = \oplus_{\text{Enc}(e_1)} T_\alpha(e_2)
\]
(22)
\[
T_\alpha(e_0 * \text{reveal}(e_1, \alpha)) = \oplus_{\text{Enc}(e_0)} T_\alpha(e_2)
\]
(23)

Fig. 8: Recursive expression transformation using \( T_\alpha \). Undefined cases (\( \bot \)) never apply for well-typed contracts.

see paper
Guarantees of ZeeStar

Correctness

...cannot violate the original contract logic

Privacy

...cannot learn more than allowed by privacy annotations

Theorem 3 (Correctness). Assume ZeeStar is instantiated with a 2k-SNARG (Def. 4). Let $C$ be the result of transforming a well-typed contract $C$. For any equivalent states $\sigma, \bar{\sigma}$ and any transaction $tx$, with overwhelming probability: running $tx$ on $C$ in starting state $\sigma$ is either rejected, or there exists a transcript $\Sigma$ for $tx$ such that $\Sigma$, $\sigma$, and public argument $\Pi$ form a proof of security state $\sigma$ resulting from validating the transaction $tx$.

Theorem 3 (Privacy). Assume ZeeStar is instantiated with a randomized well-typed encryption scheme. Let $\mathcal{A}$ be an adversary and $\mathcal{E}$ a well-typed ZeeStar contract and $\mathcal{A}$ any set of parties. Further, let $tx_{1:n}$ be any sequence of $n$ transactions, where $n$ is polynomial in the security parameter. There exists a PPT protocol $\mathcal{S}^*$ such that for any PPT adversaries $\mathcal{E}, \mathcal{E}'$, the following advantage is negligible:

$$\text{Adv}^\mathcal{E}_{\mathcal{A}}(\mathcal{S}^*(C, tx_{1:n})) - \text{Adv}^\mathcal{E}'_{\mathcal{A}}(\mathcal{S}^*(C, tx_{1:n})).$$
Implementation and Evaluation

- how to distinguish?
- restrictions?
- how to automate?
- what to prove?
- efficiency?

self-owned

foreign

homomorphic operation

plaintext operation

130a...14 + 9c8f...55 = 0aa1...bb

NIZK proof
Implementation

Implemented as an extension of zkay

Challenging to achieve efficiency

- Groth16 zk-SNARKs
- Exponential ElGamal encryption over elliptic curve
- Elliptic curve embedding

available on GitHub: eth-sri/zkay
Implementation

Implemented as an extension of zkay

Challenging to achieve efficiency

- Groth16 zk-SNARKs
- Exponential ElGamal encryption over elliptic curve
- Elliptic curve embedding

available on GitHub: eth-sri/zkay

decryption involves dlog → 32-bit values only
reflected in type system
Evaluating Example Applications

12 example contracts with example scenarios

incl. confidential Zether [FC 20]
Evaluating Example Applications

12 example contracts with example scenarios

- incl. confidential Zether [FC 20]

- tx generation time: < 55s

- feasible on commodity desktop machine

- expressive

- dominated by proof generation (57%)
Evaluating Example Applications

12 example contracts with example scenarios

- incl. confidential Zether [FC 20]

- expressive

- tx generation time: < 55s

- feasible on commodity desktop machine

- dominated by proof generation (57%)

- avg. tx gas costs: 339k gas

- comparable to existing apps (e.g., Uniswap)

- 2022-05-11: ≈ 23 USD (highly volatile)
Summary: ZeeStar

```
@datatype
owner
Token {
  mapping(address => uint) bal;
  function transfer(uint val, address to) {
    require(reveal(val <= bal[me], all));
    bal[me] = bal[me] - val;
    bal[to] = bal[to] + reveal(val, to);
  }
}
```

available on GitHub: eth-sri/zkay