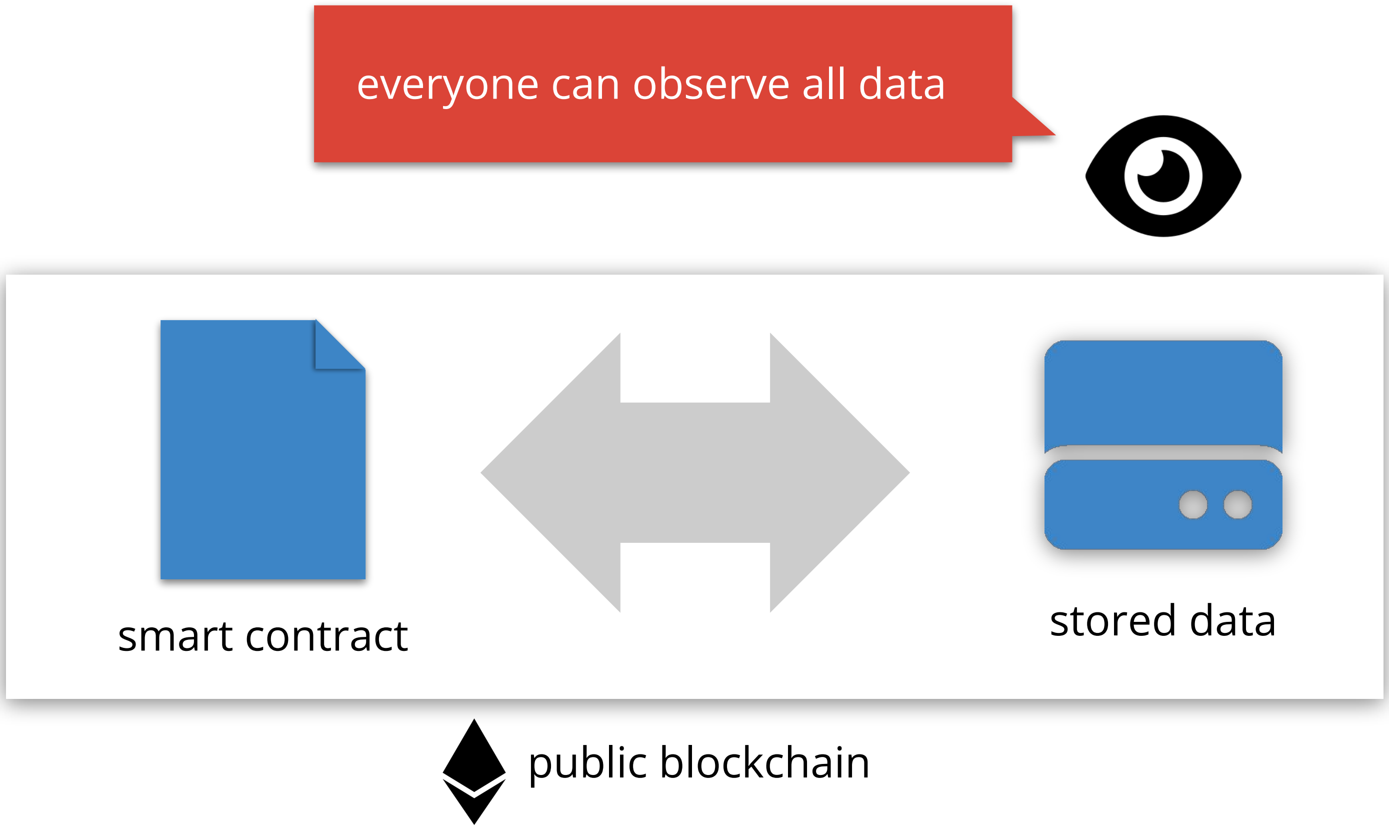
ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zero-knowledge Proofs

Samuel Steffen Benjamin Bichsel Roger Baumgartner Martin Vechev

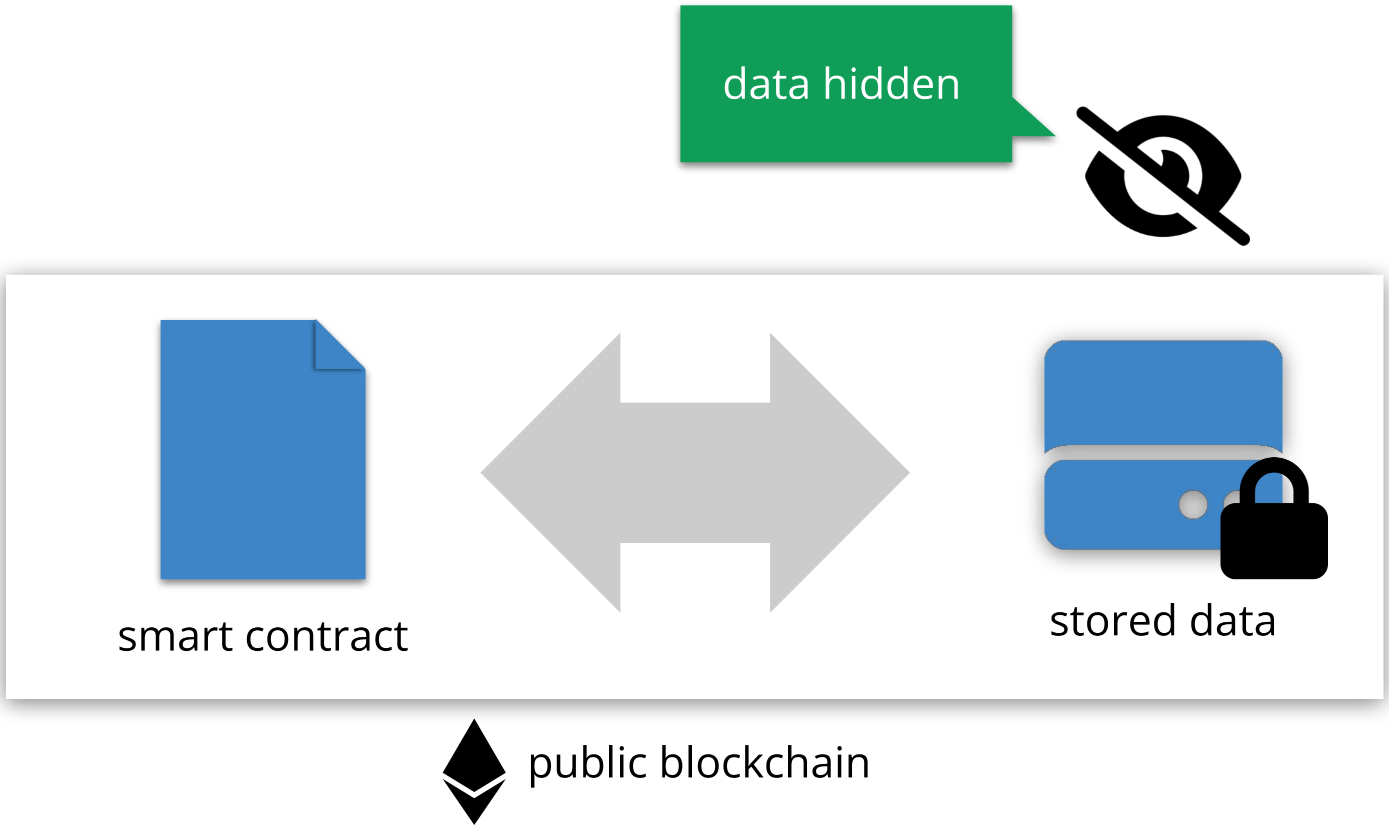
ETH Zurich, Switzerland

{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



Existing Works for Data Privacy

“30 seconds version”

Private cryptocurrencies,
Zether [FC 20], …

no general smart contracts

Existing Works for Data Privacy

"30 seconds version"

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

Existing Works for Data Privacy

"30 seconds version"

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

Existing Works for Data Privacy

"30 seconds version"

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

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

Idea and Overview

Updating Encrypted Balances

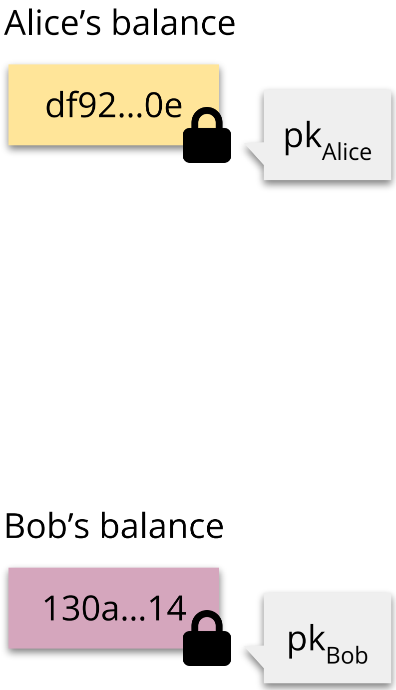
Alice's balance

42

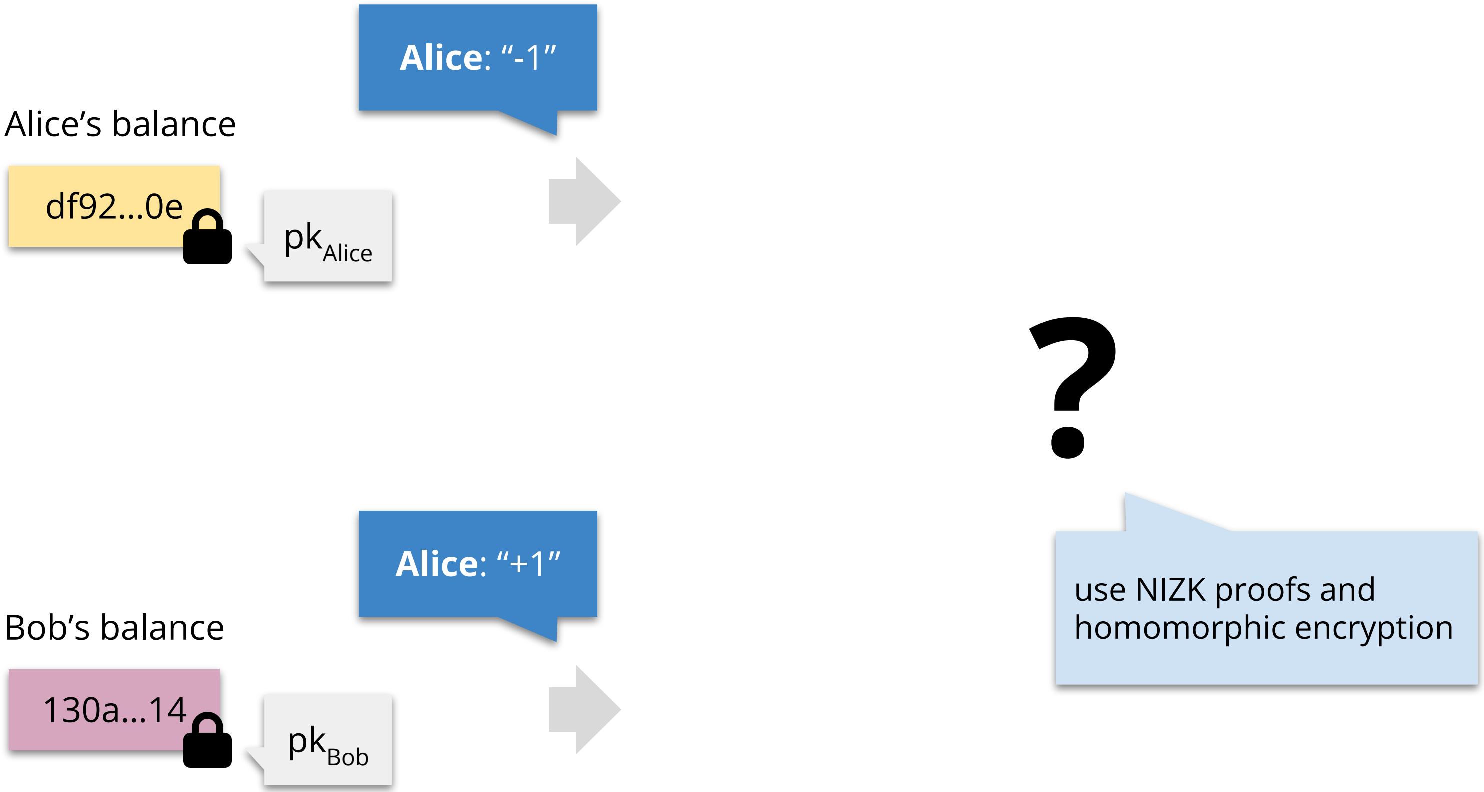
Bob's balance

56

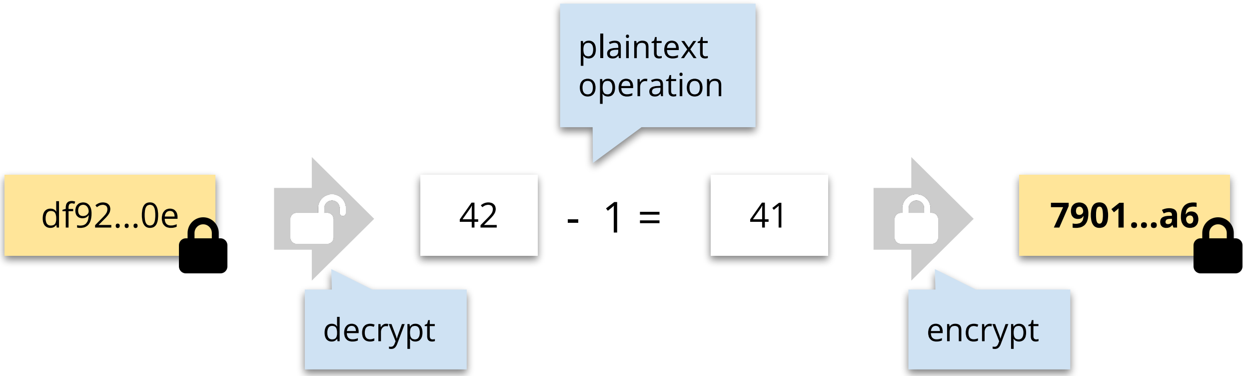
Updating Encrypted Balances



Updating Encrypted Balances

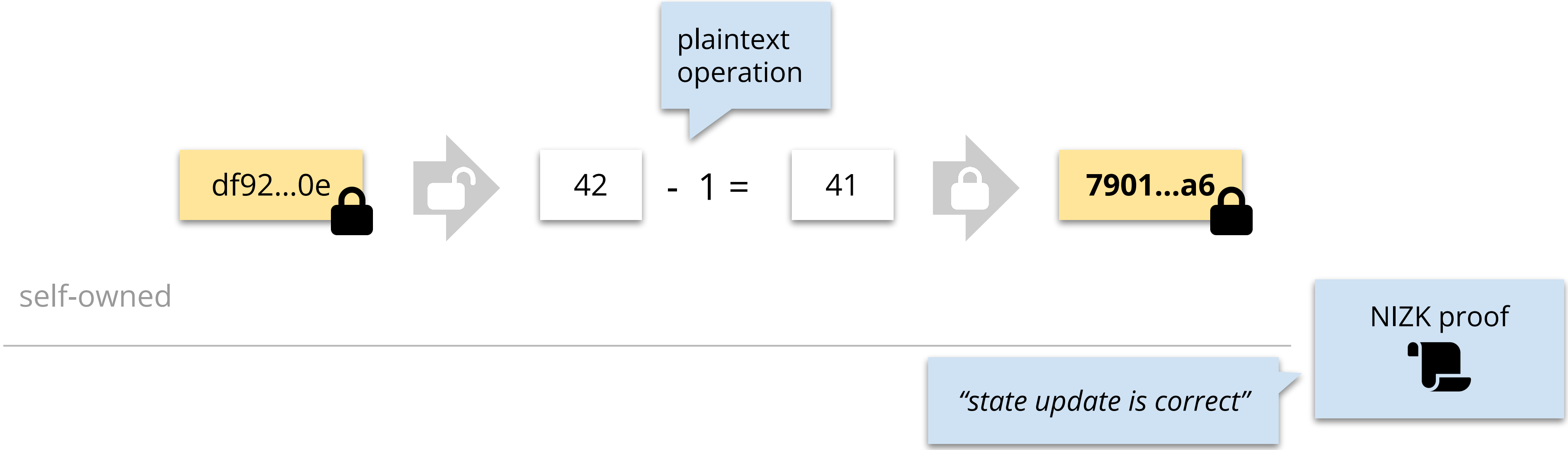


Updating Encrypted Balances

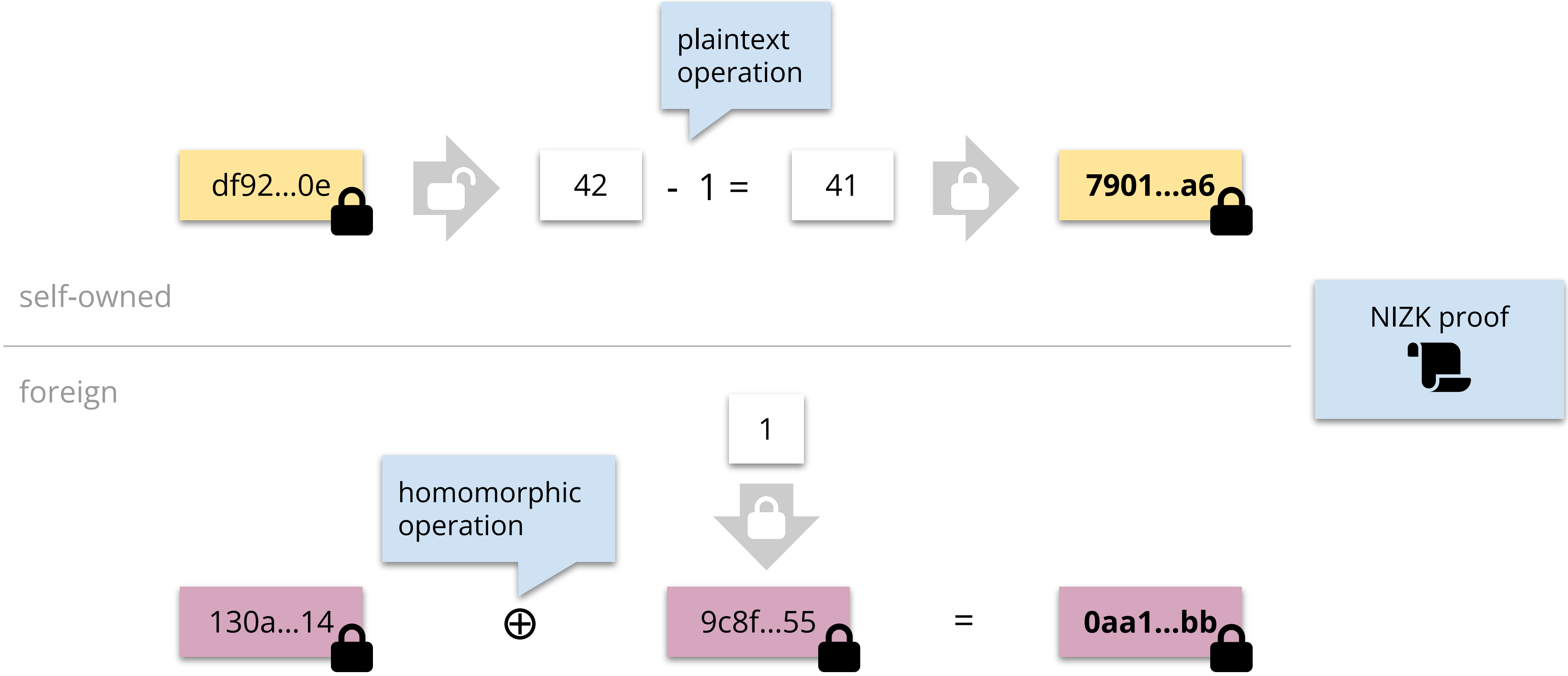


self-owned

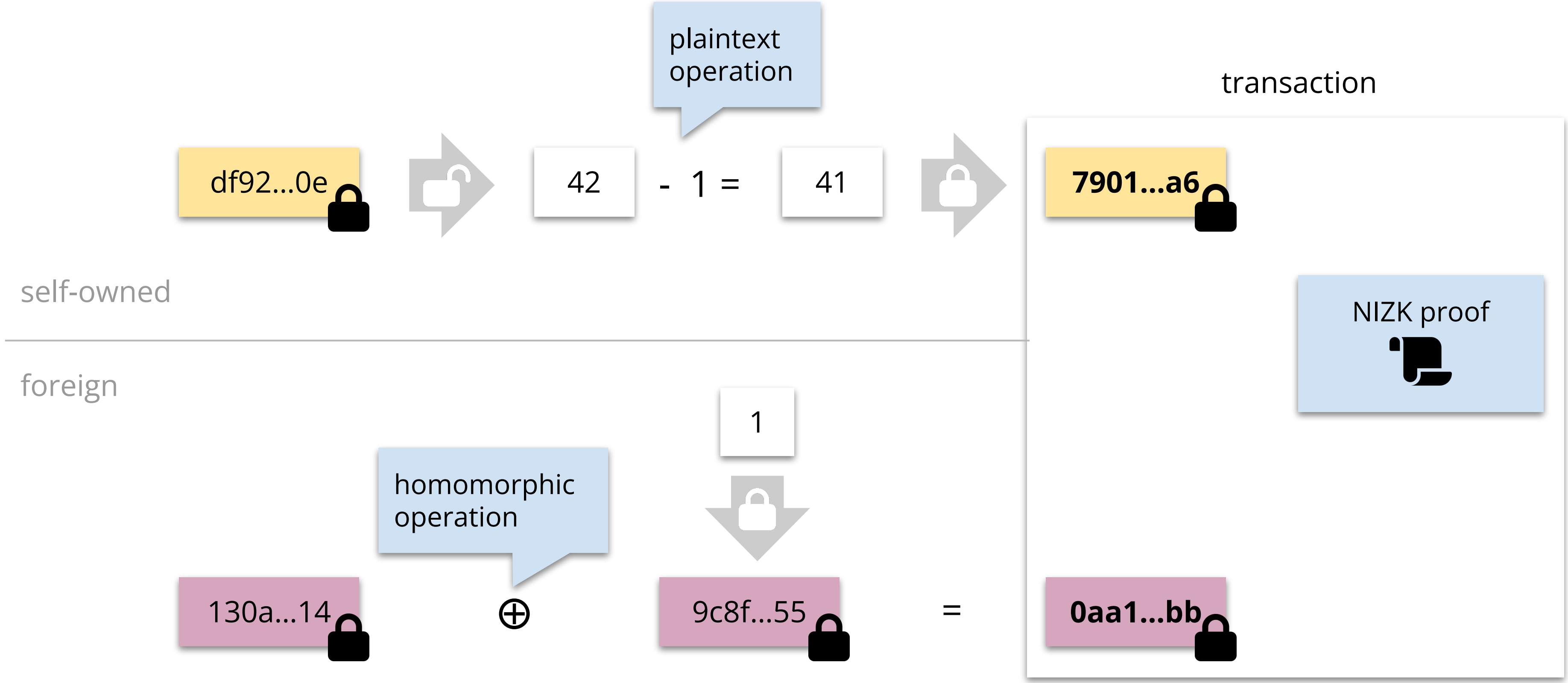
Updating Encrypted Balances



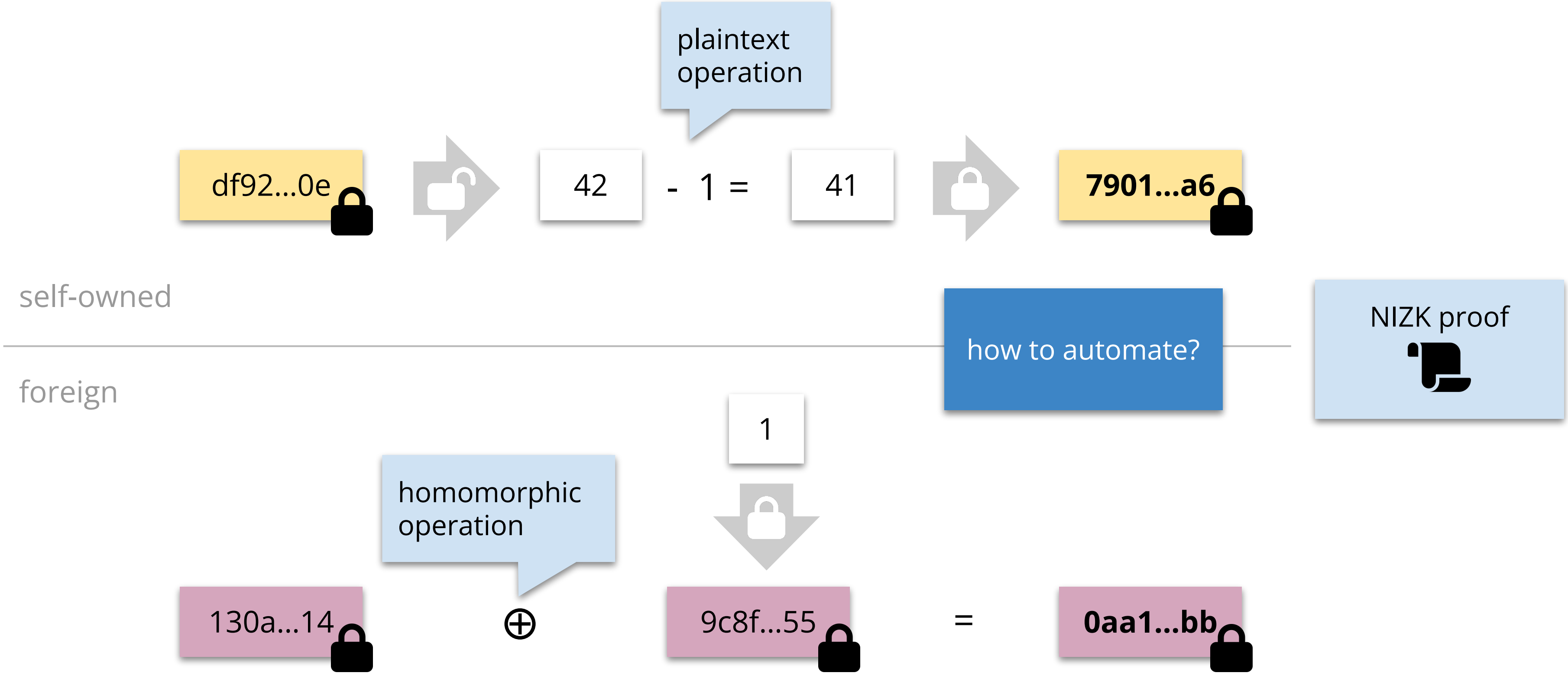
Updating Encrypted Balances



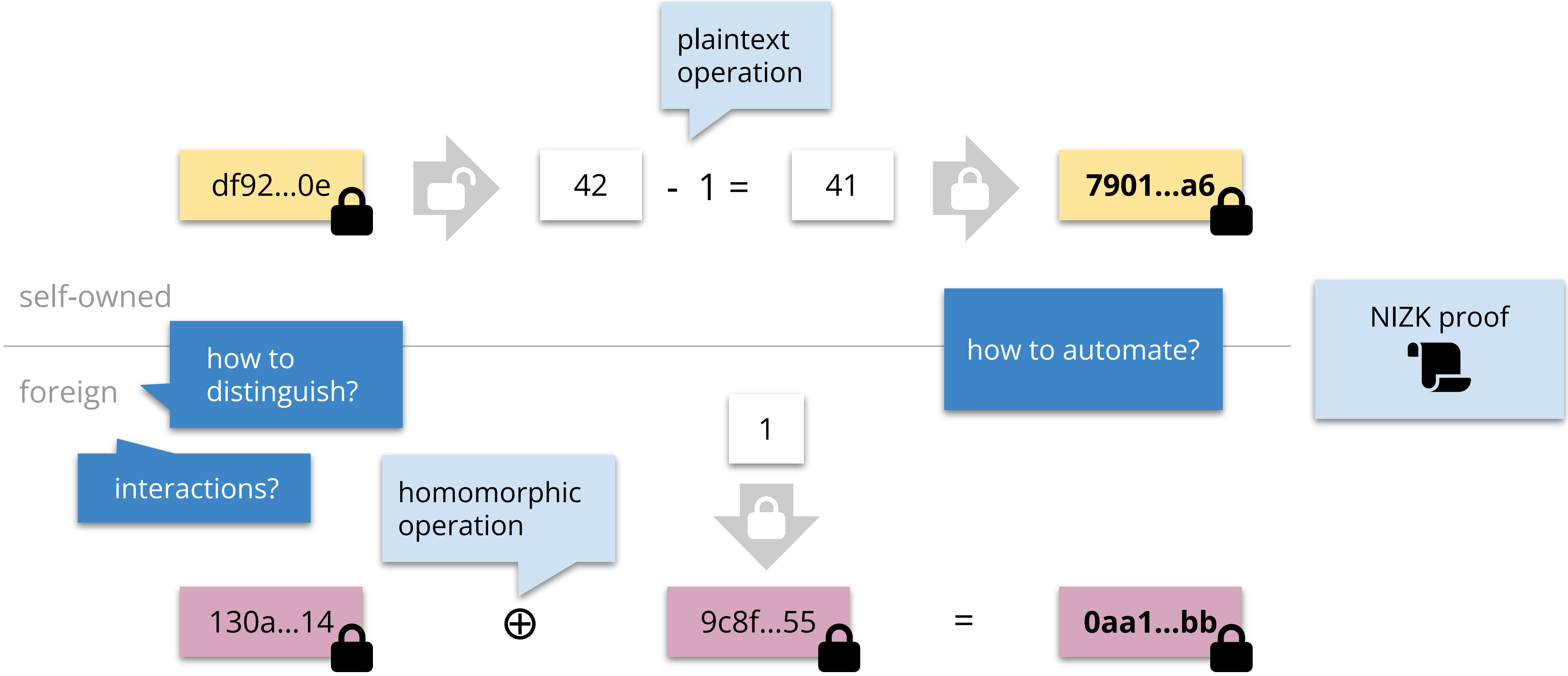
Updating Encrypted Balances



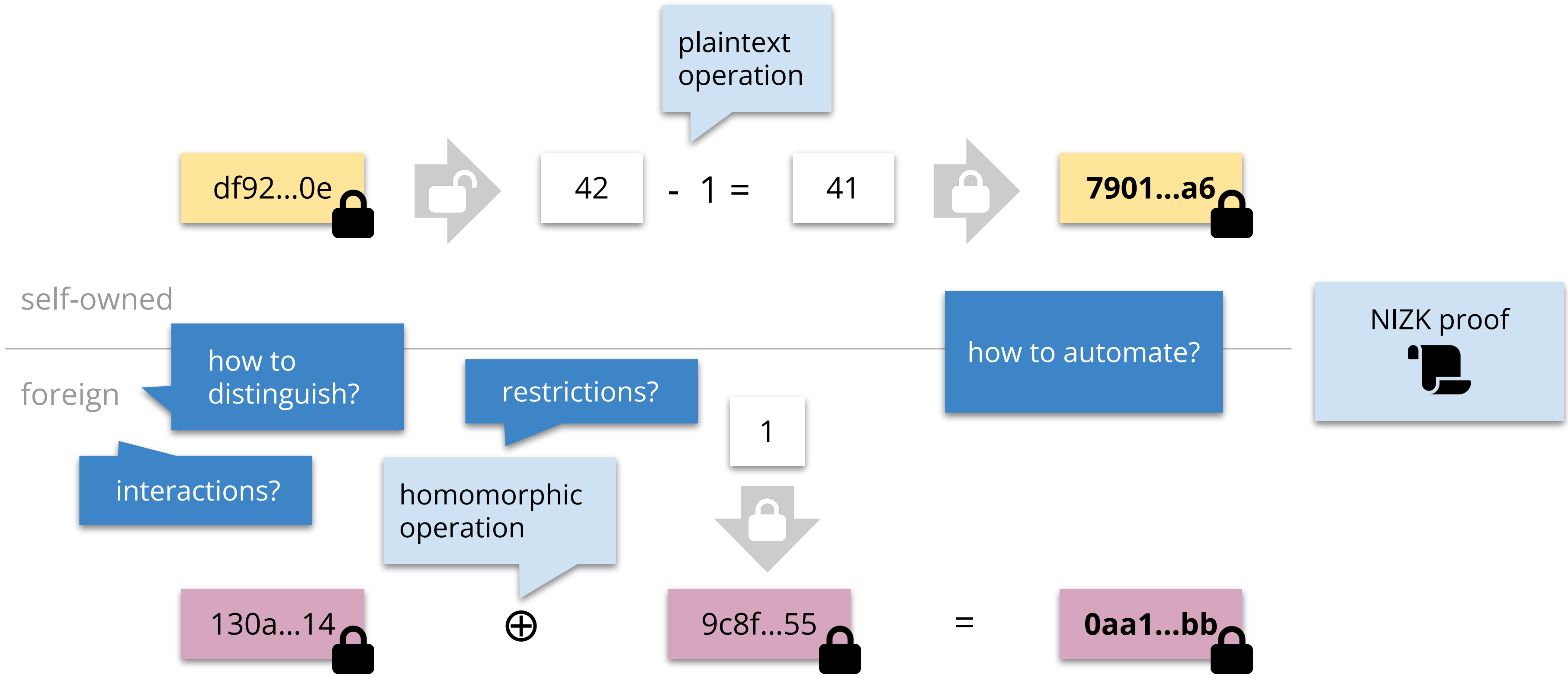
Goal: Automate



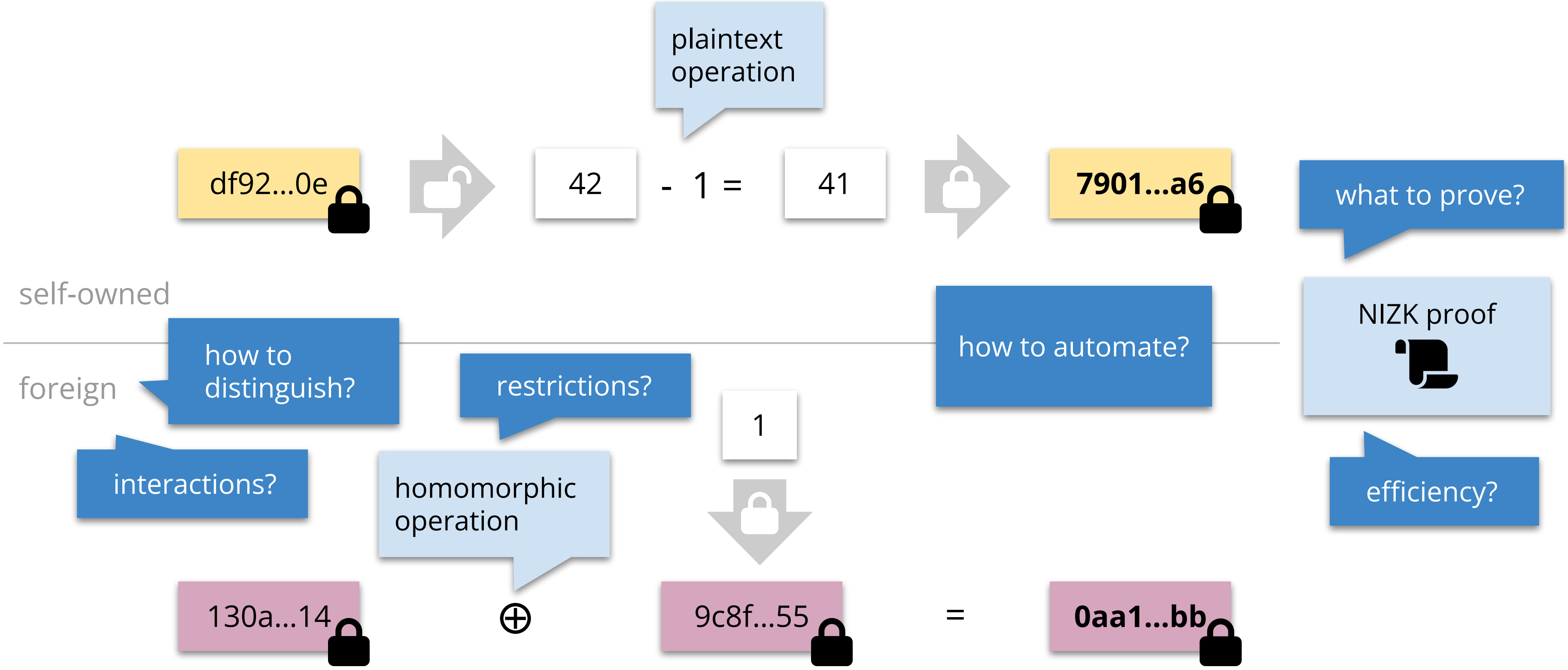
Goal: Automate



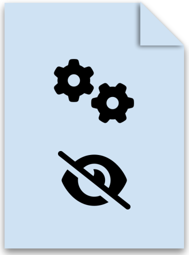
Goal: Automate



Goal: Automate



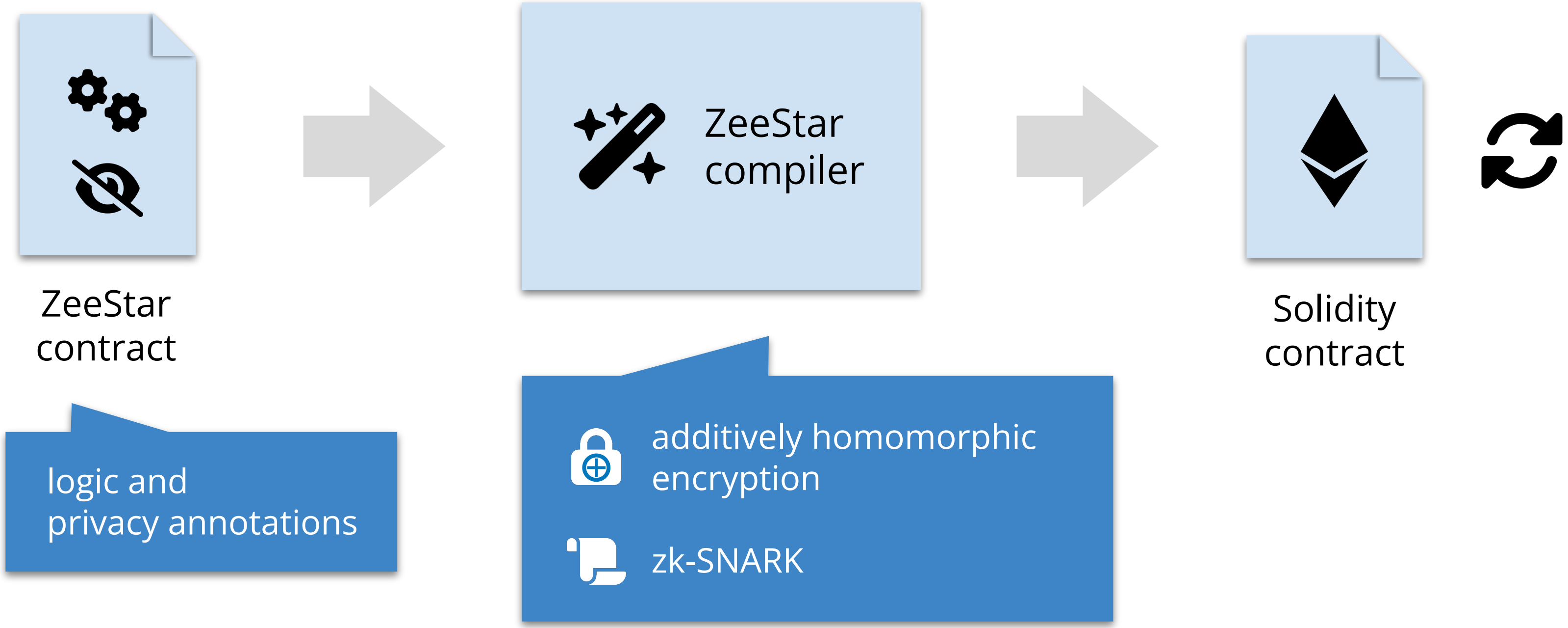
Overview: ZeeStar



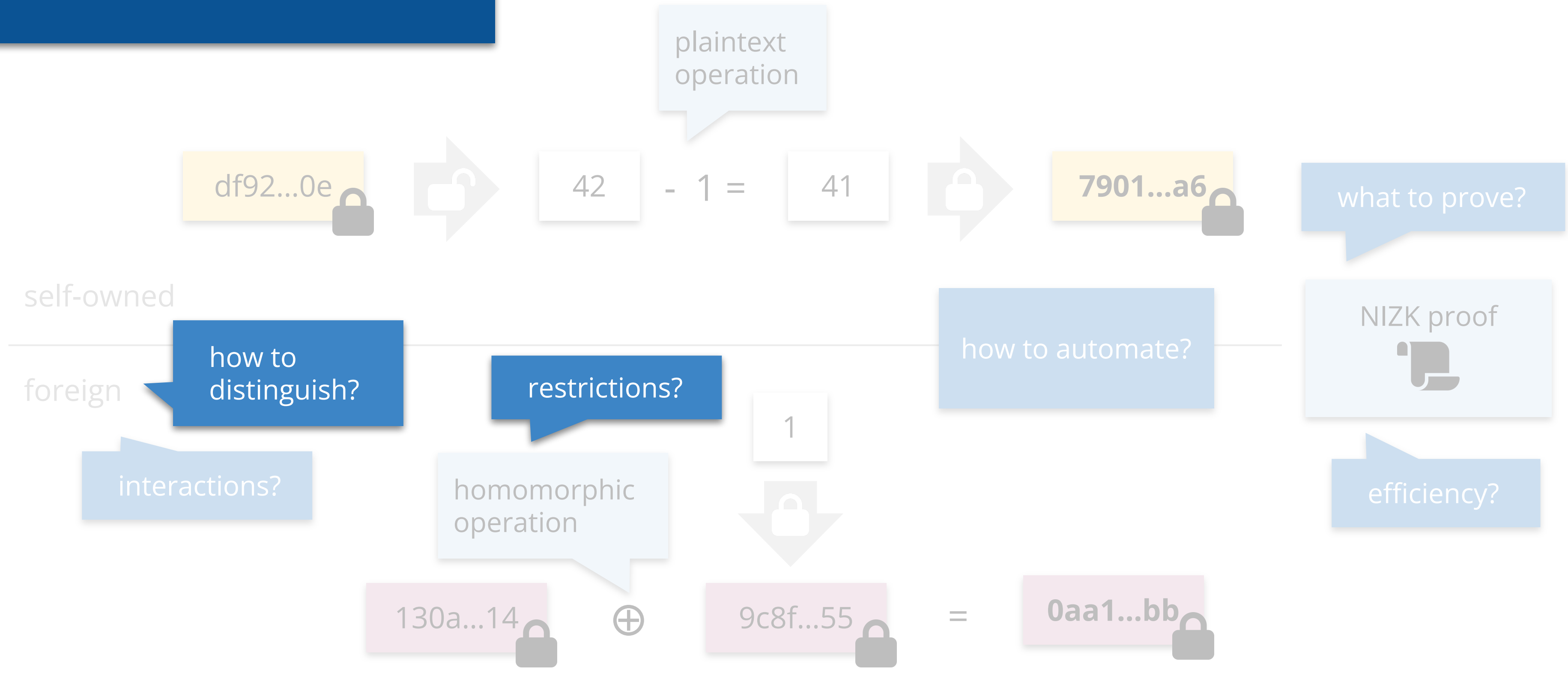
ZeeStar
contract

logic and
privacy annotations

Overview: ZeeStar



Privacy Types



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;
    }
}
```

`me = msg.sender`

Privacy Types

only party allowed to see data

`datatype@owner`

from zkay

Privacy Annotations and Types

ZeeStar

from zkay,
but adapted

```
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);
    }
}
```

Privacy Annotations and Types

from zkay,
but adapted

`bal[alice]` *owned by* `alice`

```
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);
    }
}
```

Privacy Annotations and Types

from zkay,
but adapted

self-owned (me = msg.sender)

public (@all)

```
contra
    ma                                  @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);
    }
}
```

Privacy Annotations and Types

from zkay,
but adapted

```
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);
    }
}
```

modify self-owned value

Privacy Annotations and Types

from zkay,
but adapted

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

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

modify foreign value

change privacy type
@me → @to

Privacy Annotations and Types

from zkay,
but adapted

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

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

modify foreign value

allowed
(disallowed in zkay)

change privacy type
`@me` → `@to`

Privacy Annotations and Types

from zkay,
but adapted

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

    function                        val, address to) {
        requi                       bal[me], all));
        bal[me]                     val;
        bal[to] = bal[to] + reveal(val   , to);
    }
}
```

modify foreign value

allowed
(disallowed in zkay)

change privacy type
@me → @to

type error
(cannot realize)

```
bal[bob] + bal[charlie]
```

@bob

@charlie

Privacy Annotations and Types

from zkay,
but adapted

```
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;
        ...] + reveal(val , to);

}
```

change privacy type to *public* (`@all`)

Privacy Annotations and Types

ZeeStar

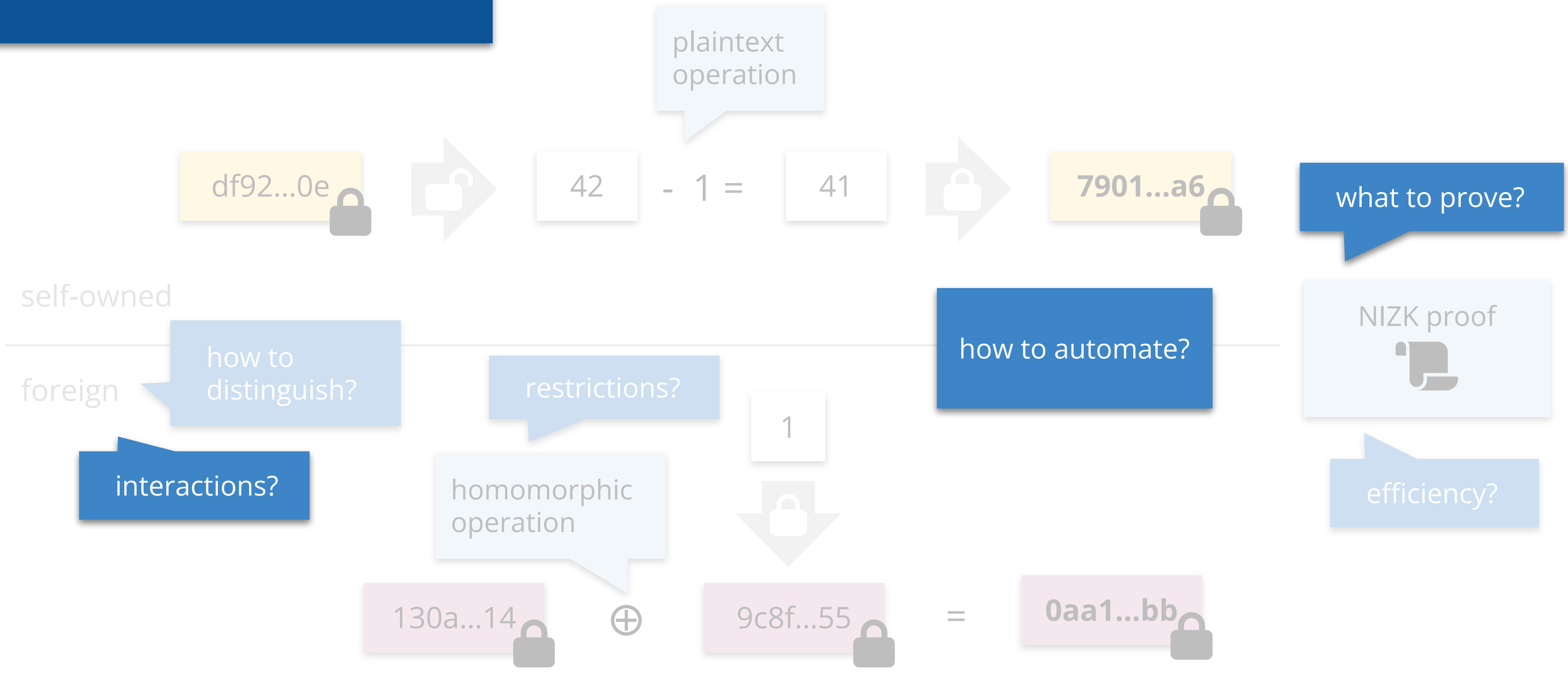
from zkay,
but adapted

```
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);
    }
}
```

no cryptographic
expertise required

Compilation



Compilation

ZeeStar

```
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);
}
```

Solidity

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

driven by privacy types

Compilation

ZeeStar

```
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);
}
```

owned by sender (@me)

Solidity

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

encrypted for owner (@me)

Compilation

```
function transfer(u            ddress to) {
  require(reveal(va            all));
  bal[me] = bal[me] - val;
  bal[to] = bal[to] + reveal(val, to);
}
```

self-owned (@me)

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

ciphertext argument

Compilation

```
function transfer(u              ddress to) {
  require(reveal(va              ], all));
  bal[me] = bal[me] - val;
  bal[to] = bal[to] + reveal(val, to);
}
```

self-owned (@me)

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

ciphertext argument

to prove:

private input (witness)

$$\text{val'} \leftarrow \text{Dec}(\text{val}, \text{sk}_{\text{me}})$$

$$\text{new\_me} == \text{Enc}(\text{Dec}(\text{bal}[\textbf{me}], \text{sk}_{\text{me}}) - \text{val'}, \text{pk}_{\text{me}}, \text{r1})$$

Compilation

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

foreign (@to)

```
functio            (...) {
  requi
  bal[me] = new_me;
  bal[to] = new_to;
  verify(proof, …);
}
```

ciphertext argument

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)
```

homomorphic addition *inside* proof (to reduce gas costs)

Compilation

revealed to public

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

plaintext argument

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

to prove:

val’ ← Dec(val, sk_{me})

ok == (val’ ≤ Dec(bal[**me**], sk_{me}))

new_me == Enc(Dec(bal[**me**], sk_{me}) - val’, pk_{me}, r1)

new_to == bal[to] ⊕ 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:

$$\texttt{val'} \leftarrow \texttt{Dec(val, sk}_{\texttt{me}}\texttt{)}$$
$$\texttt{ok == (val'} \leq \texttt{Dec(bal[}\mathbf{me}\texttt{], sk}_{\texttt{me}}\texttt{))}$$
$$\texttt{new\_me == Enc(Dec(bal[}\mathbf{me}\texttt{], sk}_{\texttt{me}}\texttt{) - val', pk}_{\texttt{me}}\texttt{, r1)}$$
$$\texttt{new\_to == bal[to]} \oplus \texttt{Enc(val', pk}_{\texttt{to}}\texttt{, r2)}$$

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);
}
```

also: *multiplication* by self-owned or public value

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

Compilation

Algorithm 1 Transforming Function Bodies

```
procedure TRANSFORM(f)
    C_f = []
    for each require(e) or id = e in the body of f do
        TRANSFORMEXPR(e, f, C_f)
    return C_f

procedure TRANSFORMEXPR(e, f, C_f)
    if e has privacy type α ≠ all then
        add new function argument arg to f
        replace e by variable arg
        add "arg ≡_α e" to C_f
    else (e is public)
        for each node e_i visited during BFS over e do
            if e_i has the form reveal(e', all) then
                add new function argument arg_i to f
                replace subtree rooted at e_i by variable arg_i
                add "arg_i ≡_all e'" to C_f
```

$$
\begin{aligned}
T_{\text{plain}}(c) &= c && (11)\\
T_{\text{plain}}(\texttt{me}) &= me && (12)\\
T_{\text{plain}}(e_1 \text{ op } e_2) &= T_{\text{plain}}(e_1) \text{ op } T_{\text{plain}}(e_2) && (13)\\
T_{\text{plain}}(\texttt{reveal}(e, \alpha)) &= T_{\text{plain}}(e) && (14)\\
T_{\text{plain}}(\text{id}) &= \begin{cases} \text{id}_{old} & \text{if id public} \\ \text{Dec}(\text{id}_{old}, sk_{me}) & \text{otherwise} \end{cases} && (15)
\end{aligned}
$$

Fig. 7: Recursive expression transformation using T_{plain}.

$$
x \equiv_\alpha e \quad \rightsquigarrow \quad \begin{cases} x = T_{\text{plain}}(e) & \text{if } \alpha = \texttt{all} \\ x = \text{Enc}(T_{\text{plain}}(e), pk_{me}, r_i) & \text{if } \alpha = \texttt{me} \\ x = T_\alpha(e) & \text{otherwise} \end{cases}
$$

Fig. 6: Transforming constraint directives to constraints.

$$
\begin{aligned}
T_\alpha(c) &= Enc_\alpha(c) && (16)\\
T_\alpha(\texttt{me}) &= Enc_\alpha(me) && (17)\\
T_\alpha(\text{id}) &= \begin{cases} \text{id}_{old} & \text{if id owned by } \alpha \\ Enc_\alpha(\text{id}_{old}) & \text{else if id public} \\ \perp & \text{otherwise} \end{cases} && (18)\\
T_\alpha(\underbrace{e_1 \text{ op } e_2}_{=:\, e}) &= \begin{cases} Enc_\alpha(T_{\text{plain}}(e)) & \text{if } e \text{ public} \\ T_\alpha(e_1) \oplus T_\alpha(e_2) & \text{else if op} = \texttt{+} \\ T_\alpha(e_1) \ominus T_\alpha(e_2) & \text{else if op} = \texttt{-} \\ \perp & \text{otherwise} \end{cases} && (19)\\
T_\alpha(\texttt{reveal}(e, \alpha')) &= \begin{cases} Enc_\alpha(T_{\text{plain}}(e)) & \text{if } \alpha = \alpha' \\ \perp & \text{otherwise} \end{cases} && (20)\\
\text{where} \quad Enc_\alpha(e) &:= \text{Enc}(e, pk_\alpha, r_i) && (21)
\end{aligned}
$$

Fig. 8: Recursive expression transformation using T_α. Undefined cases ($\perp$) never apply for well-typed contracts.

$$
\begin{aligned}
T_\alpha(e_0 * e_1) &= \oplus^{T_{\text{plain}}(e_1)} T_\alpha(e_0) && (22)\\
T_\alpha(e_0 * \texttt{reveal}(e_1, \alpha)) &= (\oplus^{T_{\text{plain}}(e_1)} T_\alpha(e_0)) && (23)\\
&\quad \oplus \text{Enc}(0, pk_\alpha, r_i)
\end{aligned}
$$

Fig. 9: Expression transformation rules for homomorphic scalar multiplication, where e_0 is foreign and e_1 is public (Eq. (22)) or self-owned (Eq. (23)). Symmetric rules omitted.

see paper

Guarantees of ZeeStar

An active adversary…

Correctness

…cannot violate the original contract logic

Privacy

…cannot learn more than allowed by privacy annotations

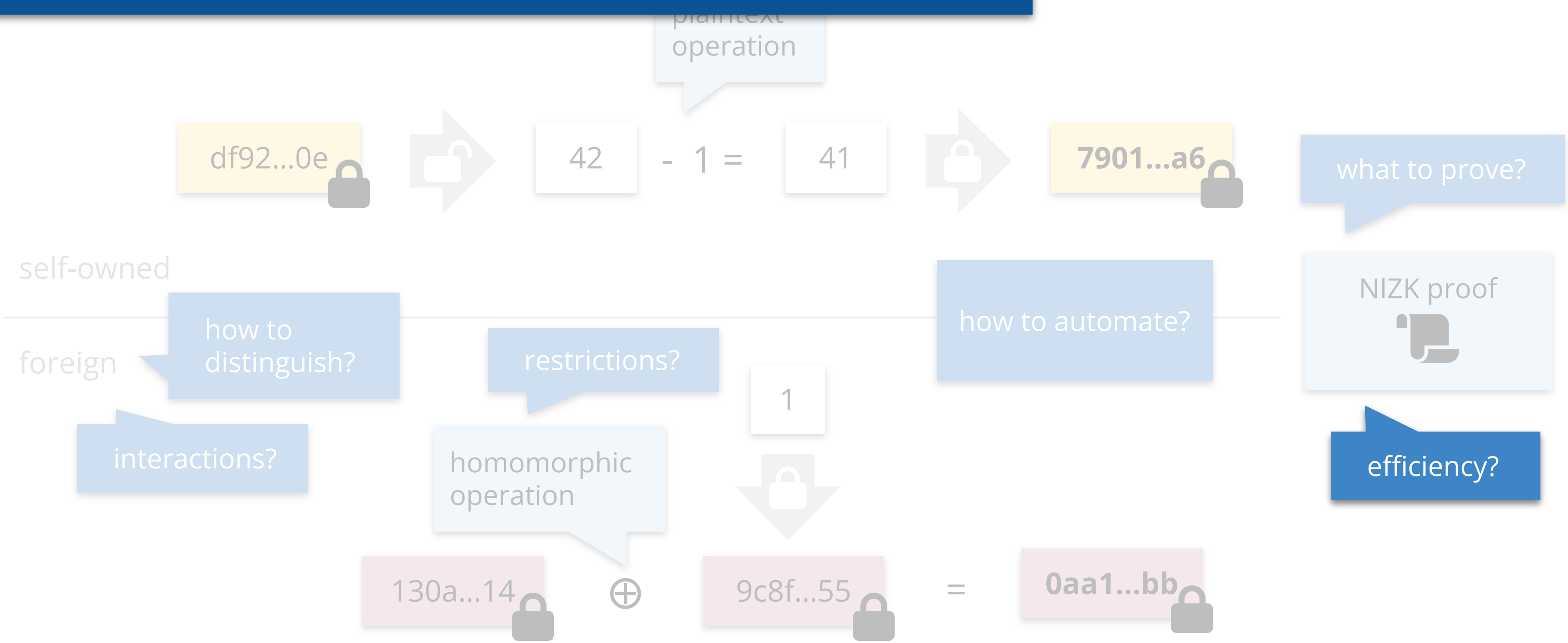
Theorem 3 (Correctness). *Assume ZeeStar is instantiated with a zk-SNARG (Def. 4). Let $\bar{C}$ be the result of transforming a well-typed contract C. For any equivalent states $\sigma, \bar{\sigma}$ and any transaction $\bar{tx}$, with overwhelming probability: running $\bar{tx}$ on $\bar{C}$ in starting state $\bar{\sigma}$ is either rejected, or there exists*
a transa … l public
argument … ng state
σ results … of $\bar{tx}$.

formalized and proven

Theorem … *ted with*
a rando … ncryption
scheme, … ell-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 $\mathcal{S}^\star$ such that for any PPT adversaries $\mathcal{E}, \mathcal{E}'$, the following advantage is negligible:

$$Adv^{\mathcal{E}'}\left(Real^{\mathcal{E}}_{\mathcal{A}}(C, tx_{1:n}),\ Sim^{\mathcal{E},\mathcal{S}^\star}_{\mathcal{A}}(C, tx_{1:n})\right).$$

Implementation and Evaluation



Implementation

available on GitHub: `eth-sri/zkay`

Implemented as an extension of zkay

Challenging to achieve efficiency

★ Groth16 zk-SNARKs

★ Exponential ElGamal encryption over elliptic curve

★ Elliptic curve embedding

Implementation

available on GitHub: eth-sri/zkay

Implemented as an extension of zkay

Challenging to achieve efficiency

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

decryption involves dlog
→ 32-bit values only

reflected in type system

Evaluating Example Applications

12 example contracts with example scenarios

expressive

incl. confidential Zether [FC 20]

Evaluating Example Applications

12 example contracts with example scenarios

expressive

incl. confidential Zether [FC 20]

tx generation time: < 55s

feasible on commodity desktop machine

dominated by proof generation (57%)

Evaluating Example Applications

12 example contracts with example scenarios

expressive

incl. confidential Zether [FC 20]

tx generation time: < 55s

feasible on commodity desktop machine

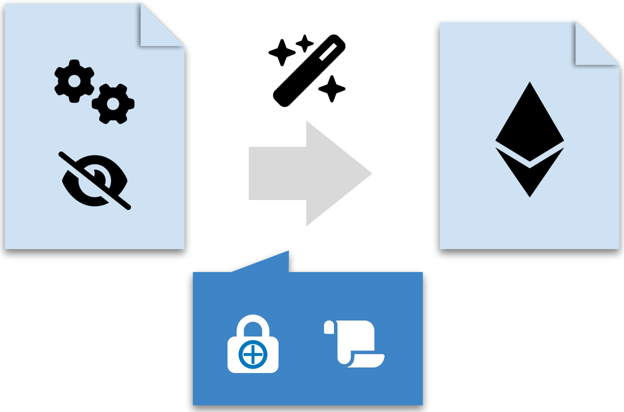
dominated by proof generation (57%)

avg. tx gas costs: 339k gas

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

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

Summary: ZeeStar



available on GitHub: eth-sri/zkay

```
datatype@owner
```

```
contract Token {
    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);
    }
}
```