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

***Abstract*—Data privacy is a key concern for smart contracts handling sensitive data. The existing work zkay addresses this concern by allowing developers without cryptographic expertise to enforce data privacy. However, while zkay avoids fundamental limitations of other private smart contract systems, it cannot express key applications that involve operations on foreign data.**

We present ZeeStar, a language and compiler allowing non-experts to instantiate private smart contracts and supporting operations on foreign data. The ZeeStar language allows developers to ergonomically specify privacy constraints using zkay's privacy annotations. The ZeeStar compiler then provably realizes these constraints by combining non-interactive zero-knowledge proofs and additively homomorphic encryption.

We implemented ZeeStar for the public blockchain Ethereum. We demonstrated its expressiveness by encoding 12 example contracts, including oblivious transfer and a private payment system like Zether. ZeeStar is practical: it prepares transactions for our contracts in at most 54.7 s, at an average cost of 339 k gas.

***Index Terms*—Privacy; Blockchain; Smart contracts; Ethereum; Programming language; Zero-knowledge proofs; Homomorphic encryption; Compilation**

I. Introduction

Modern blockchains such as Ethereum allow decentralized execution of programs (so-called smart contracts) without relying on a trusted third party. However, privacy concerns generally limit their adoption for applications processing sensitive information, such as health data [1] or voting ballots [2].

To address this, various works propose smart contract systems that respect data privacy [3]–[8]. However, these systems are subject to severe limitations. Hawk [3], Arbitrum [4], and Ekiden [5] expose significant attack surface by relying on trusted managers or hardware. Zether [6] focuses on payment systems only, which limits its applicability for general smart contracts. Finally, ZEXE [7] and smartFHE [8] require cryptographic expertise to implement new applications, inhibiting their usage by most developers.

Zkay. The recent work zkay [9, 10] has opened up a promising alternative direction. It allows non-experts to extend Solidity smart contracts by data privacy annotations indicating the owner of private values. Zkay protects these values by encrypting them for their owner and enforces updates of encrypted values to respect the smart contract logic using non-interactive zero-knowledge (NIZK) proofs. Unfortunately, a fundamental limitation of this approach is that function calls (so-called transactions) cannot operate on foreign values (i.e., values owned by parties other than the caller). This precludes zkay from expressing private variants of some of the most popular use cases of Ethereum [11], including private wallets, where coin transfers typically require increasing foreign balances.

This Work: ZeeStar. In this work, we address the expressivity restrictions of zkay by complementing it with homomorphic encryption, which allows evaluating specific operations (most importantly, addition) on foreign values.

The resulting system *ZeeStar* consists of an expressive language to specify and a compiler to automatically enforce data privacy for smart contracts. The ZeeStar language is based on zkay's privacy annotations, but additionally admits programs which operate on foreign values. The ZeeStar compiler combines NIZK proofs and additively homomorphic encryption to enable running these programs on Ethereum. By cleverly combining these two primitives, ZeeStar not only supports homomorphic addition, but also multiplication for most combinations of owners. This allows expressing complex applications such as oblivious transfer. Furthermore, ZeeStar can mix homomorphic and non-homomorphic encryption schemes and is provably private with respect to zkay's privacy notion.

Challenges. Integrating homomorphic encryption into zkay is challenging. First, homomorphic encryption and NIZK proofs have incomparable expressivity and must hence be instantiated in combination. For example, realizing a private wallet requires enforcing different ciphertexts to hold the same plaintext encrypted for different parties using a NIZK proof (§IV-A).

Second, achieving tractable prover efficiency for this combination of primitives is difficult in practice: for instance, combining Groth16 proofs [12] with Paillier encryption [13] leads to an explosion of prover memory and runtime (§VI-A).

Implementation. We implemented ZeeStar as an extension of the publicly available zkay system. Our end-to-end tool relies on exponential ElGamal encryption [14] and Groth16 NIZK proofs [12], and uses the idea of elliptic curve embedding from [7, 15] to achieve high prover efficiency. Our evaluation on 12 example contracts demonstrates that ZeeStar is expressive and its costs are comparable to popular existing applications: on average, a ZeeStar transaction costs 339 k gas (see §VII-D). Further, ZeeStar can readily express the confidential payment system Zether [6] at lower gas costs and without requiring familiarity with cryptographic primitives.

Main Contributions. Our main contributions are:

- ZeeStar, a language to specify and a compiler to automatically enforce data privacy of smart contracts (§III–§IV).
- An extension of ZeeStar to support private multiplication and mixing multiple encryption schemes (§V).
- An end-to-end implementation[1] of ZeeStar for Ethereum along with an evaluation on 12 contracts, demonstrating that ZeeStar is both expressive and practical (§VI–§VII).

II. Background

Before presenting ZeeStar, we provide a brief introduction to the two relevant cryptographic primitives.

A. Non-interactive Zero-knowledge Proofs

A non-interactive zero-knowledge (NIZK) proof [16, 17] allows a prover to demonstrate to a verifier that she knows a secret, without revealing that secret. More precisely, she can prove knowledge of a secret witness w satisfying a predicate $\phi(w; x)$ for some public value x, without revealing anything else about w other than the fact that $\phi(w; x)$ holds. We call ϕ the *proof circuit*, w the *private input*, and x the *public input*. For example, for a cyclic group G with generator g and $h \in G$, one can prove knowledge of the discrete logarithm z of h for base g using the proof circuit $\phi(z; h)$ satisfied iff $g^z = h$.

Zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) [12, 18, 19] are generic NIZK proof constructions supporting any arithmetic circuit ϕ and featuring constant-cost proof verification in the size of ϕ (plus a typically negligible linear cost in the size of x). Due to their low verification costs, zk-SNARKs are frequently used on the Ethereum blockchain [9, 10, 20].

B. Additively Homomorphic Encryption

An additively homomorphic encryption scheme allows adding the plaintexts underlying a pair of ciphertexts without knowledge of private keys. More formally, let pk_α and sk_α be the public and private key of a party α, respectively, and $\text{Enc}(x, pk_\alpha, r)$ the encryption of plaintext x under pk_α using randomness r. This scheme is additively homomorphic if there exists a function $\oplus$ on ciphertexts such that for all x, y, α, r, r':

$$\text{Enc}(x, pk_\alpha, r) \oplus \text{Enc}(y, pk_\alpha, r') = \text{Enc}(x + y, pk_\alpha, r'') \quad (1)$$

for some r'', where $\oplus$ can be efficiently evaluated without knowledge of sk_α. Note that both arguments to $\oplus$ must be encrypted under the same public key. Usually, additively homomorphic schemes also allow the homomorphic evaluation of subtraction using a function $\ominus$ defined analogously.

For example, the Paillier encryption scheme [13] is additively homomorphic in $\mathbb{Z}_n$ (that is, $+$ in Eq. (1) is addition modulo n) for an RSA modulus n, and exponential ElGamal encryption [14] over a group G is additively homomorphic in $\mathbb{Z}_{|G|}$, where $|G|$ is the order of G (see App. B).

[1]Publicly available at https://github.com/eth-sri/zkay/tree/sp2022

III. Overview

In Fig. 1, we provide an overview of ZeeStar.

Example: Private Tokens. Fig. 1a shows a ZeeStar contract modeling a wallet holding private tokens. Besides the highlighted annotations (discussed shortly) and keyword `me` (a shorthand for `msg.sender` in Solidity), the code follows a straightforward Solidity implementation. The mapping `bal` stores the number of tokens held by each individual party. The `transfer` function is used to transfer `val` tokens from the sender `me` (Line 5) to another party `to` (Line 6), after checking that the sender has sufficient funds (Line 4). For simplicity, the contract does not contain logic to initialize the balances.

Intuitively, the highlighted annotations specify the following notion of privacy: the balances of all parties must be private to the individual parties, and the number of transferred tokens must only be visible to the sender and receiver party.

Privacy Annotations and Types. To enable precise and ergonomic specification of privacy constraints, ZeeStar relies on *privacy annotations* as introduced by zkay [9]. These annotations are used to track ownership of values in a privacy type system: Data types τ (such as integers and booleans) are extended to types of the form $\tau@\alpha$, where α determines the *owner* of the expression. The value of an expression can only be seen by its owner. The owner α may be `all` (indicating the value is public), or an expression of type `address`. Expressions with owner `me` are called *self-owned*, while expressions with owner $\alpha \notin \{$`me`, `all`$\}$ are called *foreign*.

In Fig. 1a, we highlight the privacy annotations used to model the privacy notion described above. Line 2 specifies that `bal[a]` is private to the address `a`. The argument `val` of type `uint@me` (Line 3) is owned by the sender, while `to` of type `address` (a shorthand for `address@all`) is public.

In order to prevent implicit information leaks, private expressions with owner α cannot be directly assigned to variables with a different owner $\alpha' \neq \alpha$. Instead, developers can use `reveal(e, a)` to explicitly reveal a self-owned expression `e` to another owner `a`. For example, in Line 6 we reveal the transferred number of tokens `val` to the recipient `to`. This is needed because `bal[to]` is owned by `to`. To avoid implicit leaks based on access patterns, the control flow of a contract must not depend on any private values. For example, `require(`e`)` rejects the transaction (i.e., aborts and reverts it) if e evaluates to false. Thus, Line 4 publicly reveals whether the sender owns at least the number of transferred tokens.

Note that the privacy annotations only induce minimal overhead compared to existing, non-private smart contract languages such as Solidity. As discussed next, privacy is enforced automatically by ZeeStar's compiler, without requiring developers to manually instantiate cryptographic primitives. We note that zkay would reject the contract in Fig. 1a, as it cannot increase the foreign value `bal[to]` by `val` (see Line 6).

Compilation. ZeeStar compiles the input contract to a contract which is executable on Ethereum and enforces the specified privacy constraints. Fig. 1b shows a simplified version of the contract generated by ZeeStar for the token contract in Fig. 1a.

```
1  contract Token {
2    mapping(address !x => uint @x) bal;
3    function transfer(uint @me val, address to) {
4      require( reveal( val <= bal[me] , all) );
5      bal[me] = bal[me] - val;
6      bal[to] = bal[to] + reveal( val , to);
7    }
8  }
```

(a) Input ZeeStar contract with privacy annotations.

```
1  contract Token {
2    mapping(address => bin) bal;
3    function transfer(bin val, address to, bin proof,
4                      bool b, bin new_me, bin new_to) {
5      require(b);
6      bal[me] = new_me;
7      bal[to] = new_to;
8      verify_φ(proof, ...);
9    }
10 }
```

(b) Compilation output I: Solidity contract (simplified).

Public inputs:

val	encrypted `val`
bal^{me}_{new}, bal^{to}_{new}	new balances `new_me`, `new_to`
b	value of `b`
bal^{me}_{old}, bal^{to}_{old}	previous balances `bal[me]`, `bal[to]`
pk_{me}, pk_{to}	public keys of **`me`** and `to`

Private inputs (witness):

sk_{me}	private key of **`me`**
r_1, r_2	encryption randomness

Constraints:

- sk_{me} and pk_{me} form a valid key pair
- $b = (val' \leq \text{Dec}(bal^{me}_{old}, sk_{me}))$
- $bal^{me}_{new} = \text{Enc}(\text{Dec}(bal^{me}_{old}, sk_{me}) - val', pk_{me}, r_1)$
- $bal^{to}_{new} = bal^{to}_{old} \oplus \text{Enc}(val', pk_{to}, r_2)$

for $val' := \text{Dec}(val, sk_{me})$

(c) Compilation output II: Proof circuit ϕ for `transfer`.

Fig. 1: Compiling an example ZeeStar contract. The proof circuit ϕ relies on homomorphic addition $\oplus$.

In the output contract, values with owner $\alpha \neq$ **`all`** are encrypted under the public key of α using an additively homomorphic encryption scheme. Private expressions are precomputed locally (called *off-chain*) by the sender, and only published on the blockchain (*on-chain*) in encrypted form. Expressions revealed to **`all`** are additionally published in plaintext. For example, private expression `bal[me] - val` (Line 5 in Fig. 1a) is replaced by a new function argument `new_me` with ciphertext type **`bin`** (Line 6 in Fig. 1b), holding the new encrypted balance of the sender. As discussed shortly, ZeeStar uses a NIZK proof to ensure `new_me` is computed correctly. Similarly, Line 6 in Fig. 1a is transformed to Line 7 in Fig. 1b. Moreover, the revealed result of the comparison in Line 4 (Fig. 1a) is replaced by a plaintext argument `b` in Fig. 1b.

Ensuring Correctness. To ensure the function arguments `val`, `b`, `new_me`, and `new_to` are computed correctly by the sender, ZeeStar relies on both NIZK proofs and the homomorphic property of the encryption scheme. To this end, for every function, ZeeStar constructs a proof circuit ϕ enforcing correctness. Fig. 1c shows the proof circuit for `transfer`. As public inputs, ϕ takes all encrypted function arguments (`val` and new balances), revealed values (`b`), a subset of the previous state of the contract (previous balances), and the public keys of all involved parties. The private inputs consist of secrets known by the sender (most notably, her private key sk_{me}).

Intuitively, any expression involving only public and self-owned variables is computed by the sender as follows: First, decrypt any private input variables. Then, evaluate the expression on the plaintext arguments. Finally, if the expression is private, encrypt the result using the owner's public key. For example, to compute the new balance `new_me`, the sender decrypts her previous balance and the `val` argument, computes the difference, and encrypts the result under her own public key. ZeeStar collects constraints reflecting this computation in the proof circuit ϕ (Fig. 1c). Here, $\text{Dec}(x, sk)$ denotes the decryption of x using private key sk.

Leveraging Homomorphic Encryption. Because the encryption scheme is additively homomorphic, ZeeStar also allows evaluating expressions e_1 + e_2 and e_1 - e_2 for e_1, e_2 with owner $\alpha \notin \{$**`me`**, **`all`**$\}$. For example, the addition of Line 6 in Fig. 1a can be evaluated by the sender using the homomorphic operation $\oplus$. First, the sender re-encrypts the plaintext of `val` under the public key of `to` to obtain a ciphertext c. Then, the sender computes `bal[to]` $\oplus c$ to obtain `new_to`. In the proof circuit ϕ, ZeeStar ensures that c is computed correctly. Perhaps surprisingly, the operation $\oplus$ is also evaluated inside the proof circuit (see Fig. 1c). While this is not required for privacy, it leads to reduced on-chain costs (in fact, as we discuss in §VI, doing otherwise is infeasible on Ethereum). Further, as we discuss shortly, this allows for greater expressivity.

After constructing ϕ, ZeeStar inserts a proof verification statement into the output contract (see Line 8 in Fig. 1b). When calling the `transfer` function, the sender is required to generate and provide a NIZK proof for the circuit ϕ as a function argument `proof`. This is verified by the blockchain in Line 8, where the public arguments of ϕ are provided as arguments to **`verify`** (see "..."). If verification fails, the transaction is rejected and the contract state is reverted.

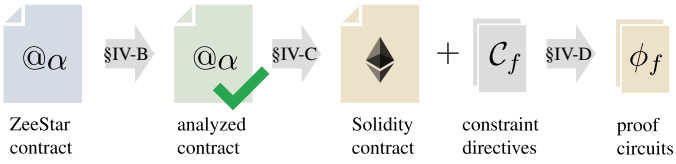


Fig. 2: Compilation steps of ZeeStar.

Extensions. The described design allows for interesting extensions. In §V-A, we describe how ZeeStar can also homomorphically evaluate multiplication for most combinations of owners. By repeated application of $\oplus$, the sender can multiply foreign values by a public natural number. Further, because $\oplus$ is evaluated inside the proof circuit, this also applies to self-owned scalars (these simply occur as plaintexts in ϕ). For example, assume `x` is owned by Alice. Bob can multiply `x` by a secret Bob-owned scalar `y`, without revealing `y` to anyone else. This opportunity is unique to the combination of NIZK proofs and additively homomorphic encryption.

In §V-B we will discuss how ZeeStar can be extended to mix homomorphic and non-homomorphic encryption schemes using suitable annotations and a modification of the type system. This is useful as practical homomorphic encryption schemes may come with restrictions (e.g., only 32-bit plaintexts), prompting the developer to only apply these selectively.

IV. Compilation

In this section, we provide a detailed description of ZeeStar. Fig. 2 visualizes the three high-level compilation steps. First, the privacy annotations of the input contract are analyzed. Then, the contract is transformed to a Solidity contract and a set of *constraint directives* $\mathcal{C}_f$ for each function f. Finally, each of these sets is transformed to a proof circuit ϕ_f.

Before describing these steps in detail (§IV-B–§IV-E), we discuss the key idea of ZeeStar's compilation process.

A. Combining NIZK Proofs and Homomorphic Encryption

Next, we discuss how combining NIZK proofs and homomorphic encryption increases expressiveness.

Incomparability of Primitives. Enforcing correctness in a ZeeStar output contract amounts to ensuring correct computation of ciphertexts (such as `new_me` in Fig. 1b). Unfortunately, the two primitives at hand are incomparable in the sense that neither is strictly more expressive than the other. While we can evaluate arbitrary expressions inside proof circuits, using NIZK proofs for correctness generally requires the prover to decrypt all input variables. For instance, the circuit in Fig. 1c decrypts the previous balance bal^{me}_{old} of the sender in order to prove correct computation of bal^{me}_{new}. In contrast, additively homomorphic encryption can be used to provide correctness guarantees "by construction," but only for addition and subtraction. For example, the sender does not need to know $sk_{\texttt{to}}$ in order to correctly update `bal[to]` in Fig. 1b.

In the example of Fig. 1, we cannot enforce correctness using only one of the primitives. Relying only on NIZK proofs and non-homomorphic encryption, the sender could not even compute the new balance `new_to`. On the other hand, only using homomorphic encryption is insufficient to guarantee correctness: First, the requirement for sufficient sender funds (Line 4 in Fig. 1a) cannot be enforced without some sort of NIZK proof. Second, while the new balance `new_me` of the sender could be updated using $\ominus$, a correct instantiation would still need to somehow enforce that the same value is removed from and added to `bal[me]` and `bal[to]`, respectively (note that the two balances are encrypted under different keys).

$$
\begin{aligned}
\alpha &::= \texttt{me} \mid \texttt{all} \mid \text{id} \\
e &::= c \mid \texttt{me} \mid \text{id} \mid e_1 \ \text{op} \ e_2 \mid \texttt{reveal}(e, \alpha) \\
S &::= S_1 ; S_2 \mid \text{id} = e \mid \texttt{require}(e) \\
\text{op} &\in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{==}, \texttt{!=}, \texttt{<=}, \texttt{<}, \texttt{\&\&}, \texttt{||}\}
\end{aligned}
$$

Fig. 3: ZeeStar core privacy types α, expressions e and statements S, where c are constants and id are variable identifiers.

Key Idea. In order to achieve high expressiveness, ZeeStar instantiates the two primitives in combination. ZeeStar's compilation is driven by privacy annotations: for each expression, ZeeStar decides which cryptographic primitive to use, based on privacy types and the actual expression. For example, because `bal[to]` is foreign, ZeeStar determines that adding `val` to it (Line 6 in Fig. 1a) requires homomorphic addition. However, because `val` is self-owned, it needs to be re-encrypted under pk_{to} in the proof circuit.

B. Privacy Type Analysis

As a first step, ZeeStar analyzes the privacy annotations in the input (see Fig. 2). Before explaining this step in an example, we first discuss the ZeeStar language in more detail.

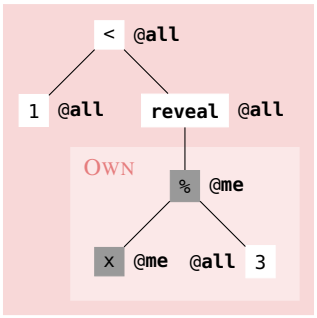
Language Fragment. In this paper, we focus on the core language fragment shown in Fig. 3. The fragment allows introducing our key ideas without cluttering the presentation.

In our fragment, function bodies consist of the statements S shown in Fig. 3. Besides sequential composition, these include assignments and **`require`** statements. The argument to **`require`** must evaluate to true for the transaction to be accepted. ZeeStar supports standard arithmetic and boolean expressions as well as a dedicated **`reveal`** expression, which is used to change the owner of a self-owned expression. Variable identifiers (id) include function arguments, contrast fields, local variables, and mapping entries, where the latter is not modeled separately for simplicity. We consider three primitive data types: booleans (**`bool`**), addresses (**`address`**), and unsigned integers (**`uint`**). In ZeeStar, variables can be self-owned (**`me`**), public (**`all`**), or owned by a public variable of type **`address`**.

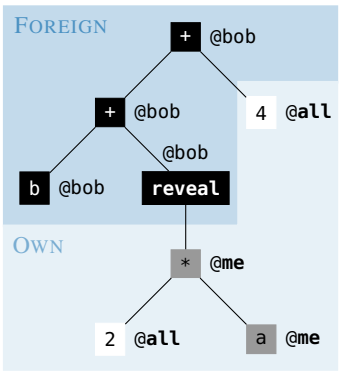
We can extend ZeeStar to other statements and expressions by following the ideas of [10]. In particular, our implementation (see §VI) accepts a much richer language based on Solidity, including non-recursive function calls (realized by inlining), if-then-else statements (realized by evaluating both branches and multiplexing), and loops. As NIZK proof circuits

```
1  contract C {
2      final address alice; uint@alice a;
3      final address bob; uint@bob b;
4      /* constructor omitted for simplicity */
5
6      function f(uint@me x) {
7          require(alice == me);
8          require( 1 < reveal(x % 3, all) );
9          b = (b + reveal(2 * a, bob)) + 4 ;
10         a = x + 1;
11     }
12 }
```

(a) Input contract.



(i) Line 8



(ii) Line 9

(b) Privacy types for the highlighted expressions in (a). The second argument to **reveal** is formally not an expression and hence not shown.

```
1 function f(bin x, uint e1, bin e2, bin e3, bin p) {
2     bin _a = a; bin _b = b;
3     require(alice == me);
4     require(1 < e1);
5     b = e2;
6     a = e3;
7     verify_φf(p, e1, e2, e3, x, _a, _b, pk(me), pk(bob));
8 }
```

$$\mathcal{C}_f = \{\texttt{e1} \equiv_{\texttt{all}} \texttt{x \% 3} \quad (2)$$
$$\texttt{e2} \equiv_{\texttt{bob}} \texttt{(b + reveal(2 * a, bob)) + 4}, \quad (3)$$
$$\texttt{e3} \equiv_{\texttt{me}} \texttt{x + 1}\} \quad (4)$$

(c) Transformed function `f` and collected constraint directives.

$$\texttt{e1} = T_{\text{plain}}(\texttt{x \% 3}) \quad (5)$$
$$\texttt{e2} = T_{\alpha}(\texttt{(b + reveal(2 * a, bob)) + 4}) \quad (6)$$
$$\texttt{e3} = \text{Enc}(T_{\text{plain}}(\texttt{x + 1}), pk_{me}, r_0) \quad (7)$$

$$\rightsquigarrow$$

$$\texttt{e1} = \text{Dec}(\texttt{x}_{old}, sk_{me}) \ \% \ 3 \quad (8)$$
$$\texttt{e2} = (\texttt{b}_{old} \oplus \text{Enc}(2 \cdot \text{Dec}(\texttt{a}_{old}, sk_{me}), pk_{bob}, r_1)) \quad (9)$$
$$\oplus \text{Enc}(4, pk_{bob}, r_2)$$
$$\texttt{e3} = \text{Enc}(\text{Dec}(\texttt{x}_{old}, sk_{me}) + 1, pk_{me}, r_0) \quad (10)$$

(d) Constructing proof circuit constraints.

Fig. 4: Running example explaining the compilation steps of ZeeStar.

have bounded size, the latter must be either free from private variables, or manually unrolled up to statically known bounds.

Running Example. To explain the compilation process of ZeeStar, we use the running example in Fig. 4. The code in Fig. 4a covers all relevant aspects of compilation but does not implement any meaningful functionality. The fields `alice` and `bob` are initialized in the constructor (not shown) and declared **`final`** to ensure they are not modified later. Like in zkay [9], this is used to prevent changing a variable's owner at runtime.

Privacy Types. ZeeStar analyzes the privacy annotations in the input contract and assigns a privacy type to each subexpression. Privacy types have two main purposes: they (i) prevent implicit information leaks, and (ii) guide the compilation by stating which expressions should be encrypted for which party. The privacy analysis ensures the privacy specification is realizable. In particular, for any well-typed contract, the subsequent compilation steps are guaranteed to succeed.

Statements. ZeeStar follows the ideas of zkay [9, 10] for analyzing statements. It requires the argument e of **`require`**(e) to be public, as the fact whether a transaction is accepted leaks the value of e. Also, for assignment statements id $= e$, the owner of both sides must be equal, or e must be public. This allows for implicitly making a public value private, but not implicitly leaking any private values.

Expressions. Privacy types of expressions are determined recursively. In Fig. 4b, we show the privacy types for the subexpressions in Line 8 and Line 9 of Fig. 4a. Constants c and the address **`me`** are public, while the privacy type of variables or mapping entries (id) is determined by their declaration. For example, in Fig. 4b-i, `x` has privacy type **`me`**, while the constant `3` is public. The **`reveal`**(e, α) expression has privacy type α, where e must be self-owned. Fig. 4b-i does not contain a node for **`all`**, because **`all`** is formally not an expression (see Fig. 3).

Fig. 5 shows the type rules for binary expressions. If both operands are public, the result is public (rule `binop-all`). For instance, in Fig. 4b-i, the inequality `<` is public as it compares public values. If one of the operands is self-owned and the other is public or self-owned, the privacy type of the result is set to **`me`** (rule `binop-me`). This is because the result depends on the private operand, so it should be kept private. For example, the `%` operation in Fig. 4b-i has privacy type **`me`**. If one operand is foreign, the only applicable operations are addition and subtraction (rule `binop-foreign`), which will later be compiled to $\oplus$ and $\ominus$, respectively. In this case, both operands must have the same owner, or one operand must be public. ZeeStar disallows mixing foreign and self-owned operands to prevent implicit leaks. If mixing is desired, developers can always **`reveal`** the self-owned operand first.

$$\frac{\Gamma \vdash e_0 : \texttt{all} \qquad \Gamma \vdash e_1 : \texttt{all}}{\Gamma \vdash e_0 \text{ op } e_1 : \texttt{all}}\ \texttt{binop-all} \qquad \frac{\Gamma \vdash e_0 : \alpha_0 \qquad \Gamma \vdash e_1 : \alpha_1 \qquad \alpha_i = \texttt{me} \qquad \alpha_{1-i} \in \{\texttt{all}, \texttt{me}\}}{\Gamma \vdash e_0 \text{ op } e_1 : \texttt{me}}\ \texttt{binop-me}$$

$$\frac{\Gamma \vdash e_0 : \alpha_0 \qquad \Gamma \vdash e_1 : \alpha_1 \qquad \alpha_i \notin \{\texttt{all}, \texttt{me}\} \qquad \alpha_{1-i} \in \{\alpha_i, \texttt{all}\} \qquad \text{op} \in \{\texttt{+}, \texttt{-}\}}{\Gamma \vdash e_0 \text{ op } e_1 : \alpha_i}\ \texttt{binop-foreign}$$

Fig. 5: Privacy type rules for binary expressions. Here, $\Gamma \vdash e : \alpha$ indicates that expression e has privacy type α.

Finding Self-owned Variables. Sometimes, the owner of a variable is syntactically different from `me` but still guaranteed to evaluate to the sender's address at runtime. For example, in Line 9 of Fig. 4a, field `a` has privacy type `alice` (by its declaration) but Line 7 ensures that `alice == me`. Like zkay, ZeeStar uses sound static analysis to find such cases. The analysis is based on a few simple, sound, but incomplete rules: For instance, a statement `require`$(a$ `==` `me`$)$ allows ZeeStar to later substitute a by `me` as long as a is not overwritten. To exploit this, ZeeStar changes the privacy type of fields to `me` whenever possible, before determining the privacy type of expressions. For example, `a` has type `@me` in Fig. 4b-ii.

C. Contract Transformation

If the input contract is well-typed, ZeeStar transforms it to a contract executable on a public blockchain and collects information required to later construct proof circuits.

Ideal World. The input contract specifies executions in an ideal world, where functions are executed according to the standard semantics of the statements and expressions in Fig. 3, but the value of an expression is only revealed to its owner. An according formal semantics can be defined analogously to [9, §4]. In the following, we use $[\![e]\!]$ to denote the plaintext value of an expression e when evaluating it in the ideal world.

Correctness. Intuitively, ZeeStar ensures that in the output contract, the value of any field is encrypted for its owner:

Theorem 1 (Correctness, informal)**.** *Let $\bar{C}$ be the output contract resulting from the compilation of a well-typed ZeeStar contract C, and z a field of C. Further, let v be the value of z in $\bar{C}$ after any sequence of transactions. Then, there exist corresponding ideal world transactions on C such that $[\![z]\!]$ in C is equivalent to v.*

Here, $[\![e]\!]$ is *equivalent to* a value v iff either e is public and $v = [\![e]\!]$, or e is owned by $\alpha \neq$ `all` and $v = \text{Enc}([\![e]\!], pk_\alpha, r)$ for some r. In App. C, we prove a more formal version of Thm. 1, which assumes that the used NIZK proof system is computationally sound (Def. 4 in App. A).

Processing a Contract. Alg. 1 describes how ZeeStar transforms a contract. This algorithm replaces publicly revealed and private expressions by new function arguments, and enforces the latter to respect equivalence as defined above.

For each function f, ZeeStar runs TRANSFORM(f), which modifies f in-place and collects a list $\mathcal{C}_f$ of *constraint directives*. A constraint directive "$x \equiv_\alpha e$" for variable x and expression e owned by α indicates that $[\![e]\!]$ must be equivalent to the value of x. Each such directive will later be transformed to a constraint in the proof circuit, thus enforcing correctness.

Algorithm 1 Transforming Function Bodies

```
1:  procedure TRANSFORM(f)
2:      C_f = [ ]
3:      for each require(e) or id = e in the body of f do
4:          TRANSFORMEXPR(e, f, C_f)
5:      return C_f
6:
7:  procedure TRANSFORMEXPR(e, f, C_f)
8:      if e has privacy type α ≠ all then
9:          add new function argument arg to f
10:         replace e by variable arg
11:         add "arg ≡_α e" to 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 reveal(e', 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 ≡_all e'" to C_f
```

For example, Fig. 4c shows the modified function `f` and the produced $\mathcal{C}_{\texttt{f}}$ when running Alg. 1 on Fig. 4a. Copies `_a`, `_b` (Line 2) and the `verify` statement (Line 7) are discussed later.

For each expression e occurring in a statement `require`(e) or id $= e$ inside the function body, Lines 3–4 (Alg. 1) run the procedure TRANSFORMEXPR$(e, f, \mathcal{C}_f)$. If e is private to α, it is replaced by a new function argument `arg` (Line 10). Further, ZeeStar adds "`arg` $\equiv_\alpha e$" to $\mathcal{C}_f$, indicating that `arg` should contain the encryption of $[\![e]\!]$ for α. In our example, the whole expression tree in Fig. 4b-ii is replaced by a new function argument `e2` with ciphertext type `bin` (Line 5 in Fig. 4c). We add Eq. (3) shown in Fig. 4c to $\mathcal{C}_{\texttt{f}}$. Line 10 in Fig. 4a is processed analogously, yielding Line 6 and Eq. (4) in Fig. 4c.

Public Expressions. Note that public expressions e may contain subexpressions of the form `reveal`$(e', \texttt{all})$, where e' is self-owned. For example, in Fig. 4a, the result of the `%` operation is revealed publicly. Hence, Alg. 1 performs a top-down tree search (for example, BFS) over public expressions e to find subtrees rooted at `reveal`$(e', \texttt{all})$ expressions. These are replaced by a new function argument, and an according constraint is added to $\mathcal{C}_f$. In our example, Line 8 in Fig. 4a is replaced by Line 4 in Fig. 4c and we record Eq. (2) in Fig. 4c.

D. Proof Circuit Construction

In the final step, for each function f, ZeeStar builds a proof circuit ϕ_f based on the previously collected $\mathcal{C}_f$.

Proof Circuit Inputs. First, ZeeStar assembles the public inputs for ϕ_f. These connect the actual values occurring in a transaction with the values in the circuit. For each "$x \equiv_\alpha e$" in $\mathcal{C}_f$, it adds public inputs x and pk_α (if $\alpha \neq$ **all**) to ϕ_f. Further, ZeeStar collects all variables id occurring in e (i.e., function arguments, contract fields, and local variables) and adds, for each id, a public input id_{old} to ϕ_f. [2] Similarly, ZeeStar adds a public input me to ϕ_f if **me** occurs in e. To simplify our explanation, we assume that function bodies are in static single assignment form: function arguments cannot be assigned to, contract fields are never read after assignment, and local variables are assigned to exactly once. By the introduction of fresh local variables, any function can be converted to this form. This ensures that constraint directives can be processed independently, and all accesses of a variable id have the same value at runtime, accessible via id_{old} in the proof circuit. In our running example, by Eqs. (2)–(4), the public proof inputs are: `e1`, `e2`, `e3`, x_{old}, a_{old}, b_{old}, pk_{me}, pk_{bob}.

The private inputs of ϕ_f include the private key sk_{me} (to decrypt self-owned values) and a list of random values r_i (see later). To enforce that sk_{me} and pk_{me} form a valid key pair, ZeeStar includes an according constraint in ϕ_f.

To pass the actual values of the public circuit inputs to ϕ_f, ZeeStar adds a proof-verification statement to the output contract. To this end, the previous values of any overwritten fields are copied at the beginning of the function. For example, in Fig. 4c, ZeeStar first copies the old values of `a` and `b` in Line 2. In Line 7, it introduces a verification statement accepting the proof `p` and all public proof inputs. Here, `pk`(α) fetches the public key of α from a public key infrastructure.

Structure of Expression Trees. We next discuss an important observation, leveraged in the rest of this work. Consider the expression tree e of a constraint directive "$x \equiv_\alpha e$" for a well-typed contract. If e contains foreign nodes, these must lie at the top of the tree and include the root node. This is enforced by the type system: foreign expressions cannot be revealed to **me** or **all** as the argument of **reveal** must be self-owned.

More precisely, we can partition the nodes of e into two sets FOREIGN and OWN, where (i) FOREIGN contains all nodes with owner $\alpha \notin \{\textbf{me}, \textbf{all}\}$, (ii) OWN contains all nodes with owner $\alpha \in \{\textbf{me}, \textbf{all}\}$, and (iii) the subgraph induced by FOREIGN is connected and, if non-empty, contains the root. Conceptually, this divides the expression tree into an upper part FOREIGN and a lower part OWN. For example, in Fig. 4b-ii, OWN contains the nodes $*$, `2`, `a`, and `4`. If the root is self-owned, then FOREIGN $= \emptyset$, as for the expression tree of constraint directive `e1` $\equiv$**all** `x % 3` (rooted at `%` in Fig. 4b-i).

[2] If id is a mapping entry of the form $e_1[e_2]$, ZeeStar also instantiates a single public input id_{old} for the entry. The mapping lookup will be performed outside ϕ_f (inside the contract), and id_{old} is assigned the value of $e_1[e_2]$. This is possible as ZeeStar's type system enforces the key e_2 to be public.

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

Fig. 6: Transforming constraint directives to constraints.

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

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

As all nodes in OWN are either self-owned or public, the sender can always compute their plaintext value. However, the value of nodes in FOREIGN is generally not known to the sender. The main idea of ZeeStar's circuit construction step is to leverage the homomorphic property of the encryption scheme for nodes in FOREIGN, and enforce correct computation of nodes in OWN by working with their plaintext values.

Transforming Expressions. We now define two recursive transformation functions T_{plain} and T_α used to build constraints for ϕ_f from expressions. The function T_{plain} is used to process nodes in OWN. It is designed such that for any $e \in$ OWN, evaluating $T_{\text{plain}}(e)$ inside the proof circuit results in $[\![e]\!]$. On the other hand, T_α targets nodes in FOREIGN and nodes in OWN whose parents are in FOREIGN. For expression e, evaluating $T_\alpha(e)$ inside the proof circuit results in the ciphertext $\text{Enc}([\![e]\!], pk_\alpha, r_i)$ for some randomness r_i.

Before discussing T_{plain} and T_α in detail, we describe how they are used. Specifically, ZeeStar transforms each constraint directive "$x \equiv_\alpha e$" in $\mathcal{C}_f$ to a constraint in ϕ_f enforcing equivalence. Depending on α, the constraint has a different form as shown in Fig. 6. If $\alpha =$ **all**, T_{plain} enforces that x holds the plaintext value of e. This ensures that self-owned values are correctly revealed by **reveal**`(`e`, all)`. If $\alpha =$ **me**, x should contain the encryption of $[\![e]\!]$ (determined using T_{plain}) under the sender's public key pk_{me} and some randomness r_i which is added to the private inputs. The third case deals with expressions for which FOREIGN is non-empty. In this case, x is owned by a party $\alpha \neq$ **me** and we leverage T_α to ensure x contains the correctly encrypted value. In our running example, based on Eqs. (2)–(4), ZeeStar adds constraints (5)–(7) in Fig. 4d to ϕ_f, where r_0 is a new private input.

Plaintext Evaluation. Fig. 7 defines the function T_{plain}. At a high level, this function decrypts any self-owned variables occurring in e and recursively evaluates the expression. The rules for constants, **me**, and binary operations are straightforward. By Eq. (14), reveal expressions are ignored (these are only used for preventing implicit leaks). Eq. (15) shows the

$$T_\alpha(c) = Enc_\alpha(c) \tag{16}$$

$$T_\alpha(\texttt{me}) = Enc_\alpha(me) \tag{17}$$

$$T_\alpha(\text{id}) = \begin{cases} \text{id}_{old} & \text{if id owned by } \alpha \\ Enc_\alpha(\text{id}_{old}) & \text{else if id public} \\ \bot & \text{otherwise} \end{cases} \tag{18}$$

$$T_\alpha(\underbrace{e_1 \text{ op } e_2}_{=:e}) = \begin{cases} Enc_\alpha(T_{\text{plain}}(e)) & \text{if } e \text{ public} \\ T_\alpha(e_1) \oplus T_\alpha(e_2) & \text{else if op} = \texttt{+} \\ T_\alpha(e_1) \ominus T_\alpha(e_2) & \text{else if op} = \texttt{-} \\ \bot & \text{otherwise} \end{cases} \tag{19}$$

$$T_\alpha(\texttt{reveal}(e, \alpha')) = \begin{cases} Enc_\alpha(T_{\text{plain}}(e)) & \text{if } \alpha = \alpha' \\ \bot & \text{otherwise} \end{cases} \tag{20}$$

$$\text{where} \quad Enc_\alpha(e) := \text{Enc}(e, pk_\alpha, r_i) \tag{21}$$

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

rule for transforming a variable id. If the variable id is public, id_{old} can be accessed directly. Otherwise, as the contract is well-typed and T_{plain} is applied only to nodes in OWN, the value is self-owned and is hence decrypted using sk_{me}. For example, in Fig. 4d, Eq. (5) is transformed to Eq. (8).

Homomorphic Evaluation. Fig. 8 defines T_α, which produces values encrypted for α. Constants and `me` are public, hence their plaintext value is encrypted under the public key of α using the function Enc_α (Eqs. (16)–(17) and Eq. (21)). Here, r_i is a new private input for ϕ_f. For foreign variables id, ZeeStar accesses id_{old}, which holds a ciphertext for α (as the contract is well-typed, T_α is never applied to private variables with owner $\neq \alpha$). If id is public, then it is encrypted for α.

For binary operations (Eq. (19)), we distinguish multiple cases. If the operation is public, then we compute its plaintext value using T_{plain} and again apply Enc_α. Private additions and subtractions are computed homomorphically: before applying $\oplus$ or $\ominus$, the arguments are recursively transformed by T_α to obtain two ciphertexts encrypted for α. Well-typed contracts do not involve other binary operations on foreign arguments.

For well-typed contracts, T_α is only applied to nodes in FOREIGN and their direct children. Hence, an expression $\texttt{reveal}(e, \alpha')$ is only reachable by T_α if $\alpha' = \alpha$, and we can apply T_{plain} and Enc_α (see Eq. (20)). Conceptually, this and all other cases introducing Enc_α provide a "bridge" between FOREIGN and OWN. Note that public expressions can be mixed with foreign expressions using `+` or `-`: for example, the constant `4` in Fig. 4b-ii is in OWN but is an argument to the root `+` in FOREIGN. Hence, Eq. (16) introduces a bridge for `4`.

In the example of Fig. 4d, Eq. (6) is transformed to Eq. (9), where r_1, r_2 are new private inputs of ϕ_f.

Transaction Transformation. To call a function f in the transformed contract, the sender needs to prepare the arguments introduced by ZeeStar. To this end, the sender selects the public arguments of ϕ_f such that ϕ_f is satisfied and generates a NIZK proof for ϕ_f (see App. D for details). Our implementation (§VI) includes a transparent interface performing these steps automatically.

E. Discussion

Privacy. ZeeStar satisfies the following notion of privacy:

Theorem 2 (Privacy, informal)**.** *Let $\bar{C}$ be the output contract resulting from the compilation of a well-typed ZeeStar contract C. An active attacker cannot learn more from real transactions on $\bar{C}$ than from the information observable in the corresponding ideal-world transactions on C (see §IV-C).*

We prove a more formal version of Thm. 2 in App. E, assuming that ZeeStar is instantiated with an IND-CPA encryption scheme and a computationally sound and perfectly zero-knowledge NIZK proof system (a weaker notion of zk-SNARK formalized as zk-SNARG in App. A). Using a hybrid argument, we show that for any probabilistic polynomial-time (PPT) adversary statically corrupting a set of parties, any sequence of real-world transactions is computationally indistinguishable from transactions simulated from information available to the adversary in the ideal world.

Limitations. ZeeStar is limited by the expressiveness of proof circuits and additively homomorphic encryption. Specifically, as proof circuits are bounded, ZeeStar contracts cannot access unbounded amounts of private memory or include unbounded loops with private operations. However, this is not a concern in practice: Due to the *block gas limit* in Ethereum, which bounds the computation of a transaction, using unbounded loops is discouraged [21]. Instead, elements of an unbounded data structure should be processed in individual transactions.

Further, foreign values can only be subject to addition or subtraction where either both operands are owned by the same party, or one operand is self-owned or public. In §V-A, we discuss how to alleviate this restriction by allowing also multiplication for most combinations of owners.

Comparison to zkay. Because ZeeStar extends zkay [9], we discuss their technical differences. Fundamentally, ZeeStar leverages NIZK proofs and homomorphic encryption (§IV-A), while zkay is limited to the former and hence strictly less expressive. The privacy annotations of ZeeStar and zkay are identical, and the privacy type analysis (§IV-B) follows the ideas of zkay for preventing implicit leaks. However, ZeeStar allows for foreign expressions disallowed in zkay and in particular treats binary operations differently (Fig. 5). Like zkay, ZeeStar replaces private and revealed expressions by new function arguments (§IV-C). However, the construction of proof circuits (§IV-D) is significantly different: while zkay only works with plaintext values, ZeeStar also tracks ciphertexts in the proof circuit (FOREIGN and T_α are unique to ZeeStar) and leverages homomorphic operations.

V. EXTENSIONS

Next, we show how ZeeStar can homomorphically evaluate multiplications for most combinations of owners (§V-A), and how different encryption schemes can be mixed (§V-B).

A. Homomorphic Multiplication by Known Scalars

Additively homomorphic encryption schemes can also be used for scalar multiplication. We can define a function $\oplus^s\ x$ which homomorphically multiplies a ciphertext x and a natural number s by homomorphically adding x to itself s times: $\oplus^s\ x := x \oplus \cdots \oplus x$. Using the double-and-add algorithm, $\oplus^s\ x$ can be computed using only $\mathcal{O}(\log s)$ applications of $\oplus$.

Multiplication by Public Scalars. The compilation process described in §IV can easily be extended to support homomorphic multiplication of foreign values by public scalars. In particular, the privacy type rule `binop-foreign` (Fig. 5) is extended to allow $e_0 * e_1$ for foreign e_0 with owner α and public e_1 (or vice-versa) and assign privacy type α to the result. When transforming expressions, ZeeStar performs this multiplication homomorphically inside the proof circuit. To this end, we extend T_α by rule (22) in Fig. 9.

For example, the contract in Fig. 4a would still compile if we replaced `+ 4` in Line 9 by `* 4`. In this case, by Eq. (22), Eq. (9) in Fig. 4d would change to

$$\texttt{e2} = \oplus^4\ (\mathrm{b}_{old} \oplus \mathrm{Enc}(2 \cdot \mathrm{Dec}(\mathrm{a}_{old}, sk_{me}), pk_{bob}, r_1)).$$

Multiplication by Self-owned Scalars. Because all homomorphic operations introduced by ZeeStar are evaluated *inside* the proof circuit, we can even extend homomorphic multiplication to self-owned scalars e_1: the plaintext value of e_1 is known to the sender and can be made available in ϕ_f using $T_{\text{plain}}(e_1)$.

Intuitively, such multiplications have the form $e_0 * e_1$, where e_0 is foreign and e_1 is self-owned. However, in order to prevent implicit leaks, ZeeStar disallows mixing self-owned and foreign operands in binary operations. Instead, ZeeStar allows expressions of the form $e_0 * \texttt{reveal}(e_1, \alpha)$ (and its symmetric variant), even though the operation $*$ is actually performed on two foreign expressions. Any other pattern $e_0 * e_1$ for foreign e_0, e_1 is not allowed because the plaintext value of e_1 cannot be guaranteed to be known by the sender.

Unfortunately, naively applying Eq. (22) to this case leads to a privacy leak. Consider Alice producing $y = \oplus^s\ x$, where x is encrypted for Bob and s is a private scalar owned by Alice. If an adversary Eve knows y and x, she can enumerate the potentially small space of possible scalars s' and find s by checking if $y = \oplus^{s'}\ x$. To prevent this attack, Alice must re-randomize y using fresh randomness before publishing y. To this end, when constructing the proof circuit, ZeeStar re-randomizes the product $z := \oplus^{T_{\text{plain}}(e_1)}\ T_\alpha(e_0)$ by homomorphically adding a freshly encrypted constant 0 to z. This is formalized in Eq. (23) of Fig. 9. Here, we assume the additional property that $\mathrm{Enc}(x, pk, r) \oplus \mathrm{Enc}(0, pk, r')$ is indistinguishable from a fresh encryption $\mathrm{Enc}(x, pk, r'')$ for any x, pk, r. This property is formalized in App. A. As we show in App. E, the above transformation preserves privacy.

Discussion. At a high level, this extension allows homomorphically multiplying two ciphertexts using an additively homomorphic encryption scheme, as long as one of these is encrypted for the sender. This is unique to the combination of NIZK proofs and additively homomorphic encryption: without the former, we could not guarantee correctness of the result. In §VII, we show how such multiplications can be used to implement 1-out-of-2 oblivious transfer.

$$T_\alpha(e_0 * e_1) = \oplus^{T_{\text{plain}}(e_1)}\ T_\alpha(e_0) \tag{22}$$

$$T_\alpha(e_0 * \texttt{reveal}(e_1, \alpha)) = (\oplus^{T_{\text{plain}}(e_1)}\ T_\alpha(e_0)) \oplus \mathrm{Enc}(0, pk_\alpha, r_i) \tag{23}$$

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.

B. Mixing Homomorphic and Non-homomorphic Schemes

In practice, homomorphic encryption schemes are often subject to restrictions. For example, exponential ElGamal encryption [14] only supports short plaintexts (≈ 32 bits; see §VI-A). Therefore, it can be useful to use non-homomorphic encryption where possible and only selectively apply homomorphic encryption where needed. We now discuss an extension of ZeeStar which allows mixing such schemes.

Homomorphism Tags. We extend ZeeStar's privacy annotations by *homomorphism tags* of the form <μ> for $\mu \in \{+, \sqcup\}$, where μ determines the homomorphic property of the encryption scheme. In particular, when declaring a variable, the developer adds a tag of the form <μ>, specifying whether the variable should be encrypted using an additively homomorphic scheme (by <+>) or a non-homomorphic scheme (by < >, or no tag). For example, in the contract of Fig. 4a, the field `a` can be encrypted non-homomorphically as it is never subject to foreign addition, by specifying the following tags:

```
2 final address alice; uint@alice a;
3 final address bob; uint@bob<+> b;
```

Encryption Schemes. Let Enc_+ and Enc be the encryption function of an additively homomorphic and non-homomorphic encryption scheme, respectively, and analogously for decryption functions Dec_+ and Dec. We modify ZeeStar's compilation process to ensure that any variable annotated as @α<μ> (for $\alpha \neq$ `all`) will be encrypted using Enc_μ at runtime.

To this end, we adapt (i) ZeeStar's proof circuit construction (§IV-D) to automatically select the appropriate encryption and decryption functions when processing a constraint directive, and (ii) ZeeStar's privacy analysis (§IV-B) to only accept contracts admitting a non-conflicting selection. Decryption is only introduced by Eq. (15), where the function Dec_μ is determined by the homomorphism tag <μ> of the variable.

Selecting the Encryption Function. Selecting the encryption function Enc_μ is more interesting. If $\alpha =$ `all` for a directive $x \equiv_\alpha e$, no encryption is needed. Otherwise, the directive must originate from Line 11 in Alg. 1 and e must therefore be the right-hand side of an assignment $l = e$. Below, we distinguish the possible cases for α.

If $\alpha =$ `me`, the second case in Fig. 6 applies. The used encryption function Enc_μ is then determined by the tag <μ>

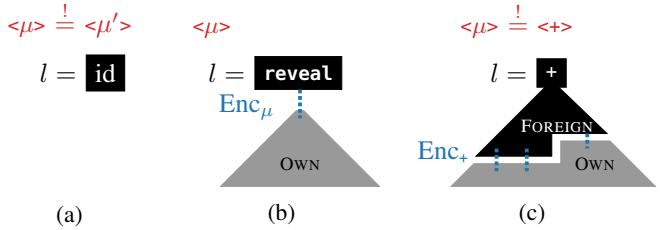


Fig. 10: Cases for determining encryption scheme.

of l. For example, for Line 10 in Fig. 4a, ZeeStar uses Enc to encrypt the result of `x + 1` because `a` is declared as `@alice`. Note that this allows for implicitly switching encryption schemes of self-owned values: an assignment $l = e$ is accepted by ZeeStar even if l is annotated `@me<+>` and e contains variables annotated as `@me` (or vice-versa). For instance, if `x` in Fig. 4a was declared as `@me<+>`, the code would still compile.

Otherwise ($\alpha \notin \{\texttt{all}, \texttt{me}\}$), we distinguish the three cases visualized in Fig. 10. If e is a foreign variable id (Fig. 10a), no encryption operation is introduced as id is already encrypted. To enable this, we adapt the privacy analysis (§IV-B) to raise a type error if the tag <μ'> of id does not match the tag <μ> of l. If e is a `reveal` expression (Fig. 10b), e is processed by Eq. (20). Then, the encryption scheme used in Enc_α is selected to match the homomorphism tag <μ> of l. Otherwise, e must be an addition or subtraction expression (Fig. 10c), to be evaluated in ϕ_f using $\oplus$ or $\ominus$ by Eq. (19). Using $\oplus$ or $\ominus$ requires their arguments to be ciphertexts under Enc_+, which recursively applies to all + and - nodes in FOREIGN. Therefore, we adapt the privacy analysis to reject private variables in FOREIGN which do not have tag <+>, and instantiate Enc_α using Enc_+ for all bridges to OWN (see Fig. 8). Further, ZeeStar ensures that the left-hand side l has tag <+>.

VI. IMPLEMENTATION

We now present our implementation of ZeeStar.

A. Efficient Cryptographic Operations

First, we discuss how encryption, decryption, and homomorphic operations can be efficiently performed within ϕ.

Expressing Proof Circuits. Verification of a zk-SNARK typically involves operations on an elliptic curve E_1 over some *base field*. E_1 determines the *scalar field* $\mathbb{F}_q$ (integers modulo q for a prime q) over which proof circuits ϕ operate. Thus, operations in ϕ must be expressed as operations over $\mathbb{F}_q$.

Problem: High Costs. Reducing ϕ to operations over $\mathbb{F}_q$ can lead to high emulation overhead for some operations (e.g., for computation over a field $\mathbb{F}_p \neq \mathbb{F}_q$), resulting in prohibitively high proof generation costs. For instance, generating a Groth16 [12] zk-SNARK for Paillier encryption [13] with 2048-bit keys requires over 256 GB of RAM—an impractical requirement for commodity desktop machines.

Solution: Curve Embedding. To address this issue, we instead leverage an encryption scheme based on an elliptic curve E_2 (discussed shortly). This allows us to rely on *curve embedding* [7, 15], which reduces prover costs for elliptic curve operations inside proof circuits. In this technique, E_1 and E_2 are carefully selected such that the base field of E_2 equals $\mathbb{F}_q$, where q is determined by E_1. This allows operations on E_2 to be evaluated natively in $\mathbb{F}_q$, without emulation overhead.

Because Ethereum currently only provides precompiled contracts for the BN254 curve [22, 23] and proof verification involves operations over E_1, we use Groth16 [12] zk-SNARKs over $E_1 = \text{BN254}$. For E_2, we use the Baby Jubjub curve [24], whose base field matches the scalar field of BN254. This choice allows for efficient cryptographic operations within ϕ.

Due to the lack of precompiled contracts, evaluating operations on E_2 on Ethereum would induce prohibitively high gas costs. However, contracts produced by ZeeStar never evaluate operations on E_2: these are only used inside proof circuits.

Setup for zk-SNARKs. Like other systems relying on Groth16 zk-SNARKs [7, 9], our implementation is subject to a circuit-specific trusted setup phase. This setup can for instance be executed using secure multi-party computation (SMC) [25].

Still, we stress that ZeeStar is fundamentally not limited to zk-SNARKs with a trusted setup. For instance, we could instantiate Bulletproofs [26] to trade the trusted setup for increased verifier complexity. Recently, several more efficient proving schemes with universal [27]–[30] or transparent setup [31] have been proposed. Once practical for Ethereum, these can likely replace the Groth16 zk-SNARKs in ZeeStar.

Homomorphic Encryption. To leverage the benefits of curve embedding, our implementation relies on exponential ElGamal encryption [14] over the Baby Jubjub curve [24]. As discussed in App. B, this scheme is additively homomorphic, provides a closed-form formula for scalar multiplication, and supports re-randomization (as required by the extension from §V-A).

In this scheme, decryption requires solving a discrete logarithm (see App. B). For small plaintext lengths k, this can be computed efficiently[3] using the baby-step giant-step algorithm [32]. However, decryption is generally intractable if k is large. Therefore, like previous works [6, 33], we restrict the plaintext to $k = 32$ bits for this encryption scheme. Longer plaintexts can still be encrypted non-homomorphically using the extension presented in §V-B. In our evaluation (§VII), a single decryption never takes longer than 7 s.

Even when restricting the plaintext size at encryption sites, the plaintext underlying a homomorphic addition $x \oplus y$ may still exceed 32 bits. Note that not knowing both x and y, there is generally no way for the sender to detect this. Like other work [6], we assume that application-specific logic is used to prevent such overflows (and similarly, underflows at 0). For example, when initializing the balances in Fig. 1a, we can use `require` statements to enforce the sum of *all* balances in `bal` to be less than 2^{32}. To make developers aware of this caveat, the type system of our implementation distinguishes integers

[3] Of course, efficient decryption requires access to the private key.

of different bit sizes (e.g., **uint32** and **uint64**), and restricts the homomorphic tag <+> to be only used with $\leq$ 32-bit integers.

B. ZeeStar for Ethereum

We implemented ZeeStar, including the extensions from §V, for Ethereum. Our Python implementation is based on the publicly available code of zkay v0.2. [4] Our end-to-end system accepts Solidity code with privacy annotations and produces (i) a contract executable on Ethereum, and (ii) a transaction interface allowing to transparently interact with ZeeStar contracts. It relies on jsnark [34] and libsnark [35] to generate zk-SNARKs. For each proof circuit, we generate a separate proof verification contract. We use solc v0.6.12 to compile Solidity code and web3 v5.19 to interact with Ethereum.

We use ElGamal encryption as described in §VI-A with 251 bit keys. For non-homomorphic encryption, we use the hybrid ECDH Chaskey cipher from zkay v0.2 [10] with 253 bit keys.

VII. Evaluation

Next, we evaluate our implementation presented in §VI. All our experiments are conducted on a machine with 32 GB of RAM and 12 CPU cores at 3.70 GHz. We use the eth-tester v0.5.0b4 backend ("Berlin" upgrade) to simulate transactions.

A. Example Contracts

We used ZeeStar to implement 12 contracts shown in Tab. I. Contracts reviews and token are homomorphic variants of the examples with the same names in [9]. Zether-confidential and zether-large are based on Zether [6] (discussed shortly). The other contracts were introduced by us. All contracts involve operations on foreign data and hence cannot be expressed by zkay [9]. Below, we discuss two contracts in more detail.

Zether Confidential Transactions. Zether [6] proposes a confidential transaction contract for Ethereum, based on additively homomorphic encryption and NIZK proofs. The contract holds encrypted balances in a table and allows sending a secret amount to another party. To prevent front-running attacks, it maintains a separate "pending" state which is used to receive currency and is periodically rolled over into the balance table. Zether allows "locking" an account to a contract such that only this contract can spend the account's balance.

Using ZeeStar, we can readily implement the idea of Zether: the contract zether-confidential implements an analogous contract in ZeeStar using just 39 lines of code (see App. F). Because ZeeStar accounts are identified by Ethereum addresses, we do not need to implement the "locking" mechanism in order to support contract-owned accounts. We note that ZeeStar leverages different primitives than Zether: ZeeStar uses Groth16 zk-SNARKs, while Zether relies on custom Σ-*Bullets* proofs. Further, the authors present an anonymous extension of Zether [6, §5.1], which we do not model because we focus on data privacy only.

[4] https://github.com/eth-sri/zkay/tree/v0.2

TABLE I: Contracts used in the evaluation. We specify the number of code lines (LoC), and whether the contracts use homomorphic addition ($\oplus$), homomorphic scalar multiplication ($\oplus^s$, see §V-A), or mix encryption schemes (<μ>, see §V-B).

No.	Name	LoC	$\oplus$	$\oplus^s$	<μ>
1	index-funds	46	•		
2	inheritance	53	•		•
3	inner-product	21	•	•	
4	member-card	25	•		
5	oblivious-transfer	19	•	•	
6	reviews	40	•		•
7	shared-prod	17	•	•	
8	token	20	•		
9	voting	40	•		
10	weighted-lottery	71	•		•
11	zether-confidential	39	•		
12	zether-large	46	•		•

Oblivious Transfer. 1-out-of-2 oblivious transfer [36] is a protocol for sending one out of two messages x_0, x_1 to a receiver. The receiver can choose i to learn x_i, without learning the other message x_{1-i} and without revealing i to the sender.

We encode such a protocol as a ZeeStar contract oblivious-transfer. First, the receiver stores his selection i in two bits $\texttt{b}_0$, $\texttt{b}_1$ with $\texttt{b}_i = 1$ and $\texttt{b}_{1-i} = 0$. The bits are owned by the receiver and enforced to be well-formed using a **require** statement. Next, the sender uses send to send messages $\texttt{x}_0$, $\texttt{x}_1$:

```
function send(uint@me x0, uint@me x1) {
    m = b0 * reveal(x0, recv) + b1 * reveal(x1, recv);
}
```

Here, recv is the address of the receiver, and m is a field owned by recv. The function uses homomorphic scalar multiplication by self-owned values $\texttt{x}_0$, $\texttt{x}_1$. The messages $\texttt{x}_i$ are both marked as being revealed to the receiver, but the receiver only learns the result of the sum: due to Eq. (19), the sum is computed inside the proof circuit. Because exactly one of the bits $\texttt{b}_i$ is 1, this is either $\texttt{x}_0$ or $\texttt{x}_1$. Note that revealing both $\texttt{x}_0$ and $\texttt{x}_1$ is unavoidable in our type system because the receiver could potentially learn both $\texttt{x}_0$ and $\texttt{x}_1$ (but not at the same time).

B. Compilation and Setup Performance

We analyze the performance of ZeeStar by compiling each example in Tab. I. In addition to the steps described in §IV, this includes, for each proof circuit, a zk-SNARK setup phase (see §VI-A). Compilation takes 66.9 s per contract on average and requires at most 3.07 GB RAM. Runtime is dominated by the setup phases (91 % of the time on average). As the setup time depends on the number and the sizes of proof circuits, compilation time varies between 26.4 s (shared-prod) and 144.1 s (zether-large). These are one-time costs per contract.

C. Transaction Generation Performance

Before a transaction is submitted to Ethereum, ZeeStar's transaction interface computes the values of the (potentially encrypted) new function arguments and generates a NIZK proof. We now evaluate the performance of this step.

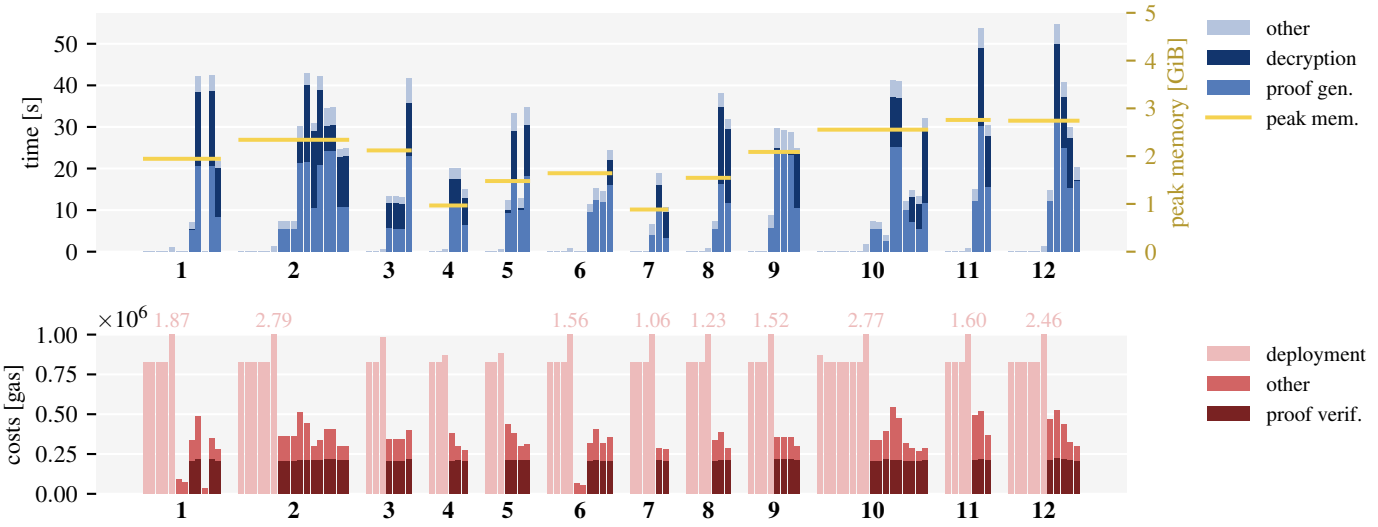


Fig. 11: Evaluation results for example scenarios. Each numbered group corresponds to a series of transactions on an example contract (see No. in Tab. I). Top: Runtime and memory for transaction generation. Bottom: Gas costs for transaction execution.

TABLE II: Number of R1CS constraints for crypto operations.

Operation	# Constraints
Encryption (Enc)	12 774
Decryption (Dec)	12 783
Re-randomization ($\oplus$ Enc$(0,\cdot,\cdot)$)	12 031
Hom. Scalar Multiplication ($\oplus^s$)	1 495
Hom. Addition ($\oplus$)	22

Scenarios. For each contract in Tab. I, we prepare a short sequence of transactions called *scenario*, which includes transactions for deploying the verification contracts (§VI-B) and executing the main contract constructor, but omits the one-time effort of deploying the public key infrastructure. For example, scenario 5 consists of three deployment transactions and two rounds of oblivious transfer (two transactions each).

Results. In Fig. 11 (top), we show the runtime and peak memory of transaction generation for all scenarios. Each bar shows the runtime of one transaction, separately indicating the runtimes of proof generation and decryption of ciphertexts (which includes solving a discrete logarithm, see §VI-A).

Generating a transaction takes at most 54.7 s and requires at most 2.8 GB of memory. The runtimes are generally dominated by proof generation (57 % of total time), whose runtime is linear in the circuit size and hence varies significantly. For some transactions (in particular, verifier deployment and most contract constructors), no proof is generated. The remaining runtime is mostly due to decryption (30 %).

Overall, ZeeStar can efficiently generate privacy-preserving transactions on commodity desktop machines.

Proof Circuit Size. In order to better understand the proof generation time, Tab. II indicates which operations in the proof circuit are most expensive. Specifically, for each cryptographic operation of the ElGamal encryption scheme presented in §VI-A, Tab. II shows the number of generated rank-1 constraint system (R1CS) constraints. As proof generation time is linear in the number of R1CS constraints, this number is a good indicator for the cost of each operation.

Encryption and decryption are the most expensive operations because these consist of relatively expensive Baby Jubjub curve point multiplications by large 256-bit scalars. Note that for decryption, we do not need to compute discrete logarithms $\log_g(x)$ inside the proof circuit: instead, we can use an additional private circuit input $z = \log_g(x)$ and assert that $x = g^z$ inside the proof circuit. Encrypting 0 can be optimized, hence re-randomization is slightly more efficient. Homomorphic scalar multiplication only requires two curve point multiplications by 32-bit scalars and hence induces fewer constraints. Finally, homomorphic addition is very efficient as it only consists of two curve point additions.

D. Transaction Execution Gas Costs

Transactions on Ethereum are subject to gas costs. We next measure these costs for the transactions generated in §VII-C.

Results. Fig. 11 (bottom) shows the gas costs for each transaction, again grouped by scenario. Deployment transactions include deployments of the verification contracts and the main contract constructors. The gas costs of such transactions are relatively high because the sender has to pay for storing the contracts' byte code. However, these are one-time costs per contract instance. For each scenario, the highest cost is induced by the main contract constructor. The overall highest costs of 2.79 M and 2.77 M are observed for inheritance and weighted-lottery, resp., which are the two largest contracts (see Tab. I).

For all non-deployment transactions, we separately indicate the costs induced by proof verification. Because the complexity of zk-SNARKs verification is essentially constant (see §II-A), these gas costs are very similar across all transactions involving a NIZK proof. The remaining costs vary between transactions, which we believe is due to the varying number

of costly storage operations. On average, a non-deployment transaction costs 339 k gas. The highest non-deployment gas cost of 544.44 k is observed for weighted-lottery.

These costs are comparable to existing applications: A transaction on Uniswap [37] (the top application on the ETH25 leaderbord[5]) frequently costs more than 250 k gas. [6]

Comparison to Zether. In [6], a Zether confidential transfer is reported to use 7 188 k gas. In contrast, the analogous ZeeStar transaction on zether-confidential only requires 521.22 k gas.

The Ethereum gas cost model has changed since the publication of [6], so we cannot directly compare these numbers. We have repeatedly tried to contact the authors to provide us the contract code or assist us with updated numbers, but we unfortunately did not receive a response. Zether relies on the Σ-*Bullets* proof system, which does not require a trusted setup but has high verifier costs. For cases where a trusted setup phase is acceptable (potentially based on SMC, see §VI-A), ZeeStar can offer significantly lower gas costs than Zether.

Monetary Costs. Transaction fees are computed by multiplying gas costs by the *gas price*, determined by supply and demand. At the time of writing, the gas price is extremely volatile: Even within a *single day*, the recommended gas price[7] fluctuates between 9 and 901 Gwei (2021-07-23). Thus, depending on the time, an average ZeeStar transaction (339 k gas) would have cost between 6.18 and 618.51 USD on this day (for 1 ETH = 2 025 USD). Hence, it is currently impossible to provide useful cost estimates for ZeeStar transactions.

The high and volatile transaction fees of Ethereum are a well-known problem [38, 39], which should be solved by the upcoming Serenity (Eth2) upgrade [40]. This will likely make ZeeStar transactions highly affordable.

VIII. Related Work

We now discuss previous work related to ZeeStar.

Smart Contract Privacy. Privacy for general smart contracts is an active area of research. In Tab. III, we compare existing approaches to ZeeStar. The table omits Kachina [43], which presents a formal security model for private smart contracts.

Arbitrum [4], Ekiden [5], and Hawk [3] expose significant attack surface by relying on trusted managers or hardware. While zkHawk [41] replaces Hawk's manager by secure multi-party computation (SMC), it requires interactive parties.

SmartFHE [8] proposes private smart contracts based on NIZK proofs and fully homomorphic encryption (FHE), where instantiating the latter at practical efficiency is known to require cryptographic expertise [44]. Further, its single-key variant requires all private inputs for a transaction to be owned by the same party. This is in contrast to ZeeStar, where the sender can combine self-owned and foreign values of multiple parties. Unfortunately, according to the authors, SmartFHE's multi-key variant is currently not practical.

[5]According to https://www.ethgasstation.info/ (accessed: 2021-07-21).

[6]For example, see (accessed on 2021-07-14): https://etherscan.io/tx/0x0894a389e86aa19ff393e921e3005867e717ef49a1296ae27f8b72f0ce0449ac

[7]"Standard" gas price according to https://www.gasnow.org/ for transactions expected to be mined within 3 minutes.

TABLE III: Tools for smart contract privacy. We indicate which tools rely on minimal trust assumptions (🛡), support Ethereum (♦), can operate on foreign values (👤), or do not require the developer to instantiate cryptographic primitives (💪).

Tool	🛡	♦	👤	💪	Remarks
Arbitrum [4]	○	○	●	●	trusted manager
Ekiden [5]	○	○	●	●	trusted hardware
Hawk [3]	○	○	●	●	trusted manager
zkHawk [41]	◐[1]	○	●	●	requires interactive parties
smartFHE [8]	●	○	◐[2]	○	
ZEXE [7]	◕[3]	○	○	◐[4]	non-standard exec. model
Zether [6, 42]	●	●	◐[5]	◐[6]	limited applications
zkay [9, 10]	◕[3]	●	○	●	
ZeeStar (ours)	◕[3]	●	◕[7]	●	

[1] At most t parties corrupted [2] Mixed owners impractical
[3] Approach proof-system-agnostic, impl. uses zk-SNARKs w/ trusted setup
[4] Manual R1CS construction [5] Addition only [6] For general applications
[7] Addition and multiplication for most combinations of owners

ZEXE [7] targets a stronger privacy notion than ZeeStar by additionally hiding the involved parties and the executed function. Unlike ZeeStar, it requires the transaction sender to decrypt all input data and hence cannot operate on foreign values. Further, it uses a non-standard execution model based on records and predicates, and contract logic needs to be manually encoded as multiple R1CS for zk-SNARKs.

While Zether [6, 42] targets payments, it can be used for a limited set of wrapper applications that rely on its interface. However, any other applications or changes to the system (e.g., enforcing a minimal amount for transactions) require manually adapting the involved cryptographic primitives.

Zkay [9, 10] (discussed in §IV-E) is a language and compiler of private contracts for Ethereum. Like ZEXE, zkay cannot operate on foreign values.

To the best of our knowledge, ZeeStar is the first tool for general private smart contracts that allows operation on foreign values, automatically instantiates cryptographic primitives, and minimizes trust assumptions. Note that while our implementation currently uses zk-SNARKs with a trusted setup phase, ZeeStar's approach is proof-system-agnostic (see also §VI-A).

Payment Privacy. A long line of work addresses privacy for cryptocurrency transactions. Here, we discuss approaches relying on NIZK proofs and/or homomorphic encryption.

Some private payment systems rely on application-specific [45, 46] or generic [15, 47]–[49] NIZK proofs. Similar to ZeeStar, several works [6, 33, 42, 50] combine homomorphic encryption and NIZK proofs to achieve payment privacy.

In contrast to all these works, ZeeStar brings data privacy to general applications beyond payments and combines NIZK proofs and homomorphic encryption automatically. ZeeStar can be used to implement private payments: for instance, it can readily express the confidential variant of Zether (see §VII-A).

Languages for SMC. Various works explore how privacy-demanding applications can be expressed in high-level languages and compiled to a combination of SMC schemes.

Some works compile standard imperative languages to a mix of SMC schemes [51, 52], while others use domain-specific languages or types to separate public and private values [53]–[55] or indicate the used SMC schemes [56]–[59].

Likewise, ZeeStar's homomorphism tags (§V-B) indicate the used encryption scheme. However, ZeeStar's privacy types indicate value ownership—a concept specific to our context. Further, these works specifically target SMC, whose execution model is fundamentally different from that of blockchains.

IX. Conclusion

We presented ZeeStar, a language and compiler for rich, private smart contracts. ZeeStar combines NIZK proofs and additively homomorphic encryption to provably realize intuitive privacy specifications on public blockchains, without requiring developers to manually instantiate cryptographic primitives. ZeeStar can express private payment systems and complex applications such as oblivious transfer. Our end-to-end implementation of ZeeStar for Ethereum is practical and its gas costs are comparable to popular existing applications.

Acknowledgment

We thank the anonymous reviewers for their constructive and detailed feedback, and the organizing committee of S&P 2022 for enabling an excellent review process.

References

[1] R. Krawiec, D. Housman, M. White, M. Filipova, F. Quarre, D. Barr, A. Nesbitt, K. Fedosova, J. Killmeyer, A. Israel, and L. Tsai, "Blockchain: Opportunities for health care," 2016, https://www2.deloitte.com/us/en/pages/public-sector/articles/blockchain-opportunities-for-health-care.html, accessed: 2021-07-02.

[2] Y. Hermstrüwer, "The limits of blockchain democracy: A transatlantic perspective on blockchain voting systems," *TTLF Working Papers*, no. 49, 2020.

[3] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts," in *2016 IEEE Symposium on Security and Privacy*. IEEE, May 2016, pp. 839–858. [Online]. Available: http://ieeexplore.ieee.org/document/7546538/

[4] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 1353–1370. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner

[5] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution," *CoRR*, vol. abs/1804.05141, 2018. [Online]. Available: http://arxiv.org/abs/1804.05141

[6] B. Bünz, S. Agrawal, M. Zamani, and D. Boneh, "Zether: Towards privacy in a smart contract world," in *Financial Cryptography and Data Security*, J. Bonneau and N. Heninger, Eds. Cham: Springer International Publishing, 2020, pp. 423–443.

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

[8] R. Solomon and G. Almashaqbeh, "smartFHE: Privacy-preserving smart contracts from fully homomorphic encryption," Cryptology ePrint Archive, Report 2021/133, 2021, https://eprint.iacr.org/2021/133.

[9] S. Steffen, B. Bichsel, M. Gersbach, N. Melchior, P. Tsankov, and M. Vechev, "zkay: Specifying and Enforcing Data Privacy in Smart Contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. London United Kingdom: ACM, Nov. 2019, pp. 1759–1776. [Online]. Available: https://dl.acm.org/doi/10.1145/3319535.3363222

[10] N. Baumann, S. Steffen, B. Bichsel, P. Tsankov, and M. Vechev, "zkay v0.2: Practical Data Privacy for Smart Contracts," Tech. Rep., 2020. [Online]. Available: https://arxiv.org/abs/2009.01020

[11] M. di Angelo and G. Salzer, "Characterizing types of smart contracts in the ethereum landscape," in *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2020, pp. 389–404. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-54455-3_28

[12] J. Groth, "On the size of pairing-based non-interactive arguments," Cryptology ePrint Archive, Report 2016/260, 2016, https://eprint.iacr.org/2016/260.

[13] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology*, ser. EUROCRYPT '99, J. Stern, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238.

[14] R. Cramer, R. Gennaro, and B. Schoenmakers, "A secure and optimally efficient multi-authority election scheme," *European Transactions on Telecommunications*, vol. 8, no. 5, pp. 481–490, Sep. 1997. [Online]. Available: http://doi.wiley.com/10.1002/ett.4460080506

[15] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, "Zcash Protocol Specification," Electric Coin Company, Tech. Rep., Mar. 2021.

[16] M. Blum, P. Feldman, and S. Micali, "Non-interactive zero-knowledge and its applications," in *Proceedings of the twentieth annual ACM symposium on Theory of computing - STOC '88*. Chicago, Illinois, United States: ACM Press, 1988, pp. 103–112. [Online]. Available: http://portal.acm.org/citation.cfm?doid=62212.62222

[17] U. Feige, D. Lapidot, and A. Shamir, "Multiple non-interactive zero knowledge proofs under general assumptions," *SIAM J. Comput.*, vol. 29, no. 1, p. 1–28, Sep. 1999. [Online]. Available: https://doi.org/10.1137/S0097539792230010

[18] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "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*. Cambridge, Massachusetts: ACM Press, 2012, pp. 326–349. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2090236.2090263

[19] J. Groth and M. Maller, "Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks," Cryptology ePrint Archive, Report 2017/540, 2017, https://eprint.iacr.org/2017/540.

[20] J. Eberhardt and S. Tai, "ZoKrates - Scalable Privacy-Preserving Off-Chain Computations," in *IEEE International Conference on Blockchain*. IEEE, 2018. [Online]. Available: http://www.ise.tu-berlin.de/fileadmin/fg308/publications/2018/2018_eberhardt_ZoKrates.pdf

[21] ConsenSys Diligence, "Known Attacks - Ethereum Smart Contract Best Practices," https://consensys.github.io/smart-contract-best-practices/known_attacks/, accessed: 2021-11-08.

[22] P. S. L. M. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," Cryptology ePrint Archive, Report 2005/133, 2005, https://eprint.iacr.org/2005/133.

[23] V. Buterin and C. Reitwiessner, "EIP-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128," 2017. [Online]. Available: https://eips.ethereum.org/EIPS/eip-197

[24] B. WhiteHat, J. Baylina, and M. Bellés, "Baby jubjub elliptic curve," 2019, Proposal. [Online]. Available: https://iden3-docs.readthedocs.io/en/latest/iden3_repos/research/publications/zkproof-standards-workshop-2/baby-jubjub/baby-jubjub.html

[25] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza, "Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs," in *SP '15*, 2015. [Online]. Available: https://doi.org/10.1109/SP.2015.25

[26] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," Cryptology ePrint Archive, Report 2017/1066, 2017, https://eprint.iacr.org/2017/1066.

[27] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, "Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings," Cryptology ePrint Archive, Report 2019/099, 2019, https://eprint.iacr.org/2019/099.

[28] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," Cryptology ePrint Archive, Report 2019/317, 2019, https://eprint.iacr.org/2019/317.

[29] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," Cryptology ePrint Archive, Report 2019/953, 2019, https://eprint.iacr.org/2019/953.

[30] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, "Marlin: Preprocessing zksnarks with universal and updatable srs," Cryptology ePrint Archive, Report 2019/1047, 2019, https://eprint.iacr.org/2019/1047.

[31] B. Bünz, B. Fisch, and A. Szepieniec, "Transparent snarks from dark compilers," Cryptology ePrint Archive, Report 2019/1229, 2019, https://eprint.iacr.org/2019/1229.

[32] D. Shanks, "Class number, a theory of factorization, and genera," in *Proc. Symp. Pure Math*, vol. 20. American Mathematical Society, 1971, pp. 415–440.

[33] P. Fauzi, S. Meiklejohn, R. Mercer, and C. Orlandi, "Quisquis: A New Design for Anonymous Cryptocurrencies," in *Advances in Cryptology – ASIACRYPT 2019*, S. D. Galbraith and S. Moriai, Eds. Cham: Springer International Publishing, 2019, pp. 649–678. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-34578-5_23

[34] A. Kosba, "jsnark: A Java library for zk-SNARK circuits," https://github.com/akosba/jsnark, accessed: 2021-08-11.

[35] SCIPR Lab and contributors, "libsnark: a C++ library for zkSNARK proofs," https://github.com/scipr-lab/libsnark, accessed: 2021-08-11.

[36] S. Even, O. Goldreich, and A. Lempel, "A randomized protocol for signing contracts," *Commun. ACM*, vol. 28, no. 6, p. 637–647, Jun. 1985. [Online]. Available: https://doi.org/10.1145/3812.3818

[37] Unsiwap Labs, "Uniswap decentralized trading protocol," https://uniswap.org/, accessed: 2021-07-14.

[38] J. Rozen, "The ridiculously high cost of Gas on Ethereum," CoinGeek Editorial, 2021, https://coingeek.com/the-ridiculously-high-cost-of-gas-on-ethereum/, accessed: 2021-07-14.

[39] Y. Faqir-Rhazoui, M.-J. Ariza-Garzón, J. Arroyo, and S. Hassan, "Effect of the Gas Price Surges on User Activity in the DAOs of the Ethereum Blockchain," in *CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3411763.3451755

[40] ethereum.org contributors, "The Eth2 vision," 2021, https://ethereum.org/en/eth2/vision/, accessed: 2021-07-14.

[41] A. Banerjee, M. Clear, and H. Tewari, "zkHawk: Practical Private Smart Contracts from MPC-based Hawk," *arXiv:2104.09180 [cs]*, May 2021, arXiv: 2104.09180. [Online]. Available: http://arxiv.org/abs/2104.09180

[42] B. E. Diamond, "Many-out-of-many proofs and applications to anonymous zether," Cryptology ePrint Archive, Report 2020/293, 2020, https://eprint.iacr.org/2020/293.

[43] T. Kerber, A. Kiayias, and M. Kohlweiss, *Kachina - Foundations of Private Smart Contracts*, 2020, published: Cryptology ePrint Archive, Report 2020/543. [Online]. Available: https://eprint.iacr.org/2020/543

[44] A. Viand, P. Jattke, and A. Hithnawi, "SoK: Fully Homomorphic Encryption Compilers," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2021, pp. 1166–1182. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00068

[45] S. Noether, "Ring signature confidential transactions for monero," Cryptology ePrint Archive, Report 2015/1098, 2015, https://eprint.iacr.org/2015/1098.

[46] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous Distributed E-Cash from Bitcoin," in *2013 IEEE Symposium on Security and Privacy*. Berkeley, CA: IEEE, May 2013, pp. 397–411. [Online]. Available: http://ieeexplore.ieee.org/document/6547123/

[47] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized Anonymous Payments from Bitcoin," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 459–474. [Online]. Available: https://ieeexplore.ieee.org/document/6956581

[48] Z. Guan, Z. Wan, Y. Yang, Y. Zhou, and B. Huang, "BlockMaze: An Efficient Privacy-Preserving Account-Model Blockchain Based on zk-SNARKs," *IEEE Transactions on Dependable and Secure Computing*, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9200775

[49] A. Rondelet and M. Zajac, "ZETH: On Integrating Zerocash on Ethereum," *arXiv:1904.00905 [cs]*, Apr. 2019, arXiv: 1904.00905. [Online]. Available: http://arxiv.org/abs/1904.00905

[50] S. Ma, Y. Deng, D. He, J. Zhang, and X. Xie, "An Efficient NIZK Scheme for Privacy-Preserving Transactions Over Account-Model Blockchain," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 2, pp. 641–651, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/8968397

[51] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, "Hycc: Compilation of hybrid protocols for practical secure computation," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 847–861. [Online]. Available: https://doi.org/10.1145/3243734.3243786

[52] M. Ishaq, A. L. Milanova, and V. Zikas, "Efficient mpc via program analysis: A framework for efficient optimal mixing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1539–1556. [Online]. Available: https://doi.org/10.1145/3319535.3339818

[53] J. D. Nielsen and M. I. Schwartzbach, "A domain-specific programming language for secure multiparty computation," in *PLAS'07*, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1255329.1255333

[54] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations," in *2014 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2014, pp. 655–670. [Online]. Available: http://ieeexplore.ieee.org/document/6956593/

[55] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A Programming Framework for Secure Computation," in *SP '15*, May 2015. [Online]. Available: https://ieeexplore.ieee.org/document/7163036

[56] D. Bogdanov, P. Laud, and J. Randmets, "Domain-Polymorphic Programming of Privacy-Preserving Applications," in *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*, ser. PLAS'14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 53–65. [Online]. Available: https://doi.org/10.1145/2637113.2637119

[57] D. Demmler, T. Schneider, and M. Zohner, "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation," in *NDSS '15*, 2015. [Online]. Available: http://dx.doi.org/10.14722/ndss.2015.23113

[58] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "Aby2.0: Improved mixed-protocol secure two-party computation," Cryptology ePrint Archive, Report 2020/1225, 2020, https://eprint.iacr.org/2020/1225.

[59] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "Tasty: Tool for automating secure two-party computations," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 451–462. [Online]. Available: https://doi.org/10.1145/1866307.1866358

[60] Y. Tsiounis and M. Yung, "On the security of elgamal based encryption," in *Public Key Cryptography*, H. Imai and Y. Zheng, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 117–134.

APPENDIX

A. Security Definitions

Below, we summarize security definitions used throughout the appendix. By *negligible*, we mean negligible in the (implicit) security parameters of the cryptographic primitives.

Definition 1 (Advantage)**.** *For probabilistic algorithms D_0, D_1 and a probabilistic polynomial-time (PPT) algorithm $\mathcal{E}$, the* advantage *$Adv^{\mathcal{E}}(D_0, D_1)$ is defined as*

$$Adv^{\mathcal{E}}(D_0, D_1) := \big|Pr[\mathcal{E}(x) = 1 : x \leftarrow D_0] - Pr[\mathcal{E}(x) = 1 : x \leftarrow D_1]\big|.$$

Definition 2 (IND-CPA)**.** *A public-key encryption scheme with encryption function Enc is* IND-CPA *if, for any PPT adversary $\mathcal{E}$ and any two messages m_0, m_1 of equal length, the following advantage is negligible:*

$$Adv^{\mathcal{E}}(F(0, m_0, m_1),\ F(1, m_0, m_1)),$$

where $F(i, m_0, m_1)$ generates a fresh public key pk and uniform randomness r to return $(m_0, m_1, pk, Enc(m_i, pk, r))$.

Definition 3 (Randomizability)**.** *An additively homomorphic public-key encryption scheme with encryption function Enc is* randomizable *if, for any PPT adversary $\mathcal{E}$, any m and any r*

$$Adv^{\mathcal{E}}(F(m, r),\ G(m, r)) = 0,$$

where F generates a fresh public key pk and uniform randomness r' to return $(m, r, pk, Enc(m, pk, r'))$, and G generates a fresh public key pk and uniform randomness r' to return $(m, r, pk, Enc(m, pk, r) \oplus Enc(0, pk, r'))$.

Definition 4 (zk-SNARG [12])**.** *A* zero-knowledge succinct non-interactive argument system *is a tuple $(Setup, Prove, Verify)$ of PPT algorithms, where $Setup(\phi)$ returns a common reference string (CRS) Σ and trapdoor τ, $Prove(\Sigma, \phi, x, w)$ returns a proof π for the circuit $\phi(w; x)$, $Verify(\Sigma, \phi, x, \pi)$ returns 1 iff π is considered valid, and:*

- Succinctness. *The size of π is polynomial in the security parameter λ, and Verify runs in time polynomial in $\lambda + |x|$.*
- Perfect completeness. *For any ϕ, x, w such that $\phi(w; x)$ holds, it is*

$$Pr\left[Verify(\Sigma, \phi, x, \pi) = 1 : \begin{matrix}(\Sigma, \tau) \leftarrow Setup(\phi) \\ \pi \leftarrow Prove(\Sigma, \phi, x, w)\end{matrix}\right] = 1.$$

- Computational soundness. *For any PPT adversary $\mathcal{E}$ and any ϕ, the following is negligible:*

$$Pr\left[\begin{matrix}Verify(\Sigma, \phi, x, \pi) = 1 \\ \wedge\ \nexists w.\ \phi(w; x)\ holds\end{matrix} : \begin{matrix}(\Sigma, \tau) \leftarrow Setup(\phi) \\ (x, \pi) \leftarrow \mathcal{E}(\Sigma, \phi)\end{matrix}\right].$$

- Perfect zero-knowledge. *There exists a PPT simulator SimProof such that for any PPT adversary $\mathcal{E}$ and any ϕ, x, w s.t. $\phi(w; x)$ holds, $Adv^{\mathcal{E}}(F, G) = 0$, where*
  - *F runs $(\Sigma, \tau) \leftarrow Setup(\phi)$, $\pi \leftarrow Prove(\Sigma, \phi, x, w)$ to return $(\Sigma, \tau, \phi, x, \pi)$*
  - *G runs $(\Sigma, \tau) \leftarrow Setup(\phi)$, $\pi \leftarrow SimProof(\Sigma, \tau, \phi, x)$ to return $(\Sigma, \tau, \phi, x, \pi)$*

B. Properties of Exponential ElGamal Encryption

The ElGamal encryption scheme with messages in the exponent [14] is defined over a cyclic group G. It is IND-CPA (Def. 2), assuming the decisional Diffie-Hellman assumption holds in G [60].

For group G with order $|G|$ and generator g, the private key sk_α of a party α is selected uniformly at random from $\{1, \ldots, |G| - 1\}$ and its public key is derived as $pk_\alpha = g^{sk_\alpha}$.

Let k with $2^k \leq |G|$ be the maximal message bit length. For uniformly chosen randomness $r \in \{0, \ldots, |G| - 1\}$, the encryption of a message $m \in \{0, \ldots, 2^k - 1\}$ is

$$\text{Enc}(m, pk, r) := (g^r,\ g^m \cdot pk^r).$$

Decryption of ciphertext (c_1, c_2) using the private key sk is

$$\text{Dec}((c_1, c_2), sk) := \log_g(c_2 \cdot c_1^{|G| - sk}),$$

where $\log_g$ denotes the discrete logarithm to the base g.

Defining $(c_1, c_2) \oplus (d_1, d_2) := (c_1 \cdot d_1,\ c_2 \cdot d_2)$, this scheme is additively homomorphic:

$$\begin{aligned}&\text{Enc}(x, pk, r) \oplus \text{Enc}(y, pk, r') = (g^{r+r'},\ g^{x+y} \cdot pk^{r+r'}) \\ &= \text{Enc}(x + y,\ pk,\ r + r'),\end{aligned} \tag{24}$$

where $+$ is addition modulo $|G|$. Homomorphic subtraction can be defined as $(c_1, c_2) \ominus (d_1, d_2) := (c_1, c_2) \oplus (d_1^{-1}, d_2^{-1})$.

Homomorphic scalar multiplication by a natural number s can be defined in closed-form as $\oplus^s\ (c_1, c_2) := (c_1^s, c_2^s)$, which can be efficiently computed using the well-known double-and-add algorithm involving $\mathcal{O}(\log s)$ applications of $\oplus$. By homomorphically adding a freshly encrypted constant 0, an existing ciphertext can be re-randomized:

$$\begin{aligned}&\text{Enc}(m, pk, r) \oplus \text{Enc}(0, pk, r') = (g^{r+r'},\ g^m \cdot pk^{r+r'}) \\ &= (g^{r''},\ g^m \cdot pk^{r''}) = \text{Enc}(m, pk, r'')\end{aligned}$$

for $r'' := r + r' \mod |G|$. Because r' is a uniformly random number in $\{0, \ldots, |G| - 1\}$, so is r'' and the result is perfectly indistinguishable from a fresh encryption of m. Hence, this scheme is randomizable according to Def. 3.

C. Correctness of ZeeStar

In Thm. 3, we provide a more formal version of Thm. 1. Here, a state σ is *equivalent to* a state $\bar{\sigma}$ if both states include the same contract fields, and the value of every field z in σ is equivalent to the value of z in $\bar{\sigma}$. By an inductive argument, for any polynomial-length sequence of transactions on $\bar{C}$ in the empty starting state, with overwhelming probability there exists a corresponding sequence of transactions on C.

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 $t\bar{x}$, with overwhelming probability: running $t\bar{x}$ on $\bar{C}$ in starting state $\bar{\sigma}$ is either rejected, or there exists a transaction tx for the same function, sender and public arguments as $t\bar{x}$ such that running tx on C in starting state σ results in state σ' equivalent to the output state $\bar{\sigma}'$ of $t\bar{x}$.*

Proof sketch. First, we prove that sk_{me} and pk_{me} inside the proof circuit belong to the transaction sender with overwhelming probability. As the blockchain authenticates the sender of a transaction (e.g., via a signature in Ethereum), the contract $\bar{C}$ ensures that pk_{me}, which is passed as a public input for proof verification, belongs to the original sender. If the transaction is accepted, the zk-SNARG is successfully verified by an honest verifier. Therefore, by the computational soundness property, a private input sk_{me} such that ϕ is satisfied must exist with overwhelming probability. As sk_{me} is enforced to correspond to pk_{me} in ϕ (e.g., see Fig. 1c), it is guaranteed to belong to the original sender with overwhelming probability.

Given that sk_{me} and pk_{me} are correct, the correctness of state updates follows from the computational soundness of the zk-SNARG and inductive reasoning on the transformation rules (Figs. 6–9), which ensure correctness by construction via the constraint directives $x \equiv_\alpha e$. □

Malleability. Note that proof malleability is not a problem for correctness, as the argument above only relies on the soundness property. Further, impersonation attacks are prevented, assuming the ledger requires all transactions to be signed by sk_{me} corresponding to the sender's public key pk_{me}. Then, an adversary α trying to submit a tampered proof π' must sign it with sk_α. Thus, because the ledger forwards pk_α as a public input to π', π' is only accepted if it enforces correctness with respect to pk_α—thus defeating the purpose of tampering with the proof in the first place.

D. Transaction Transformation

Let C be an input ZeeStar contract, and $\bar{C}$ the transformed output contract. In order to call a function f on $\bar{C}$ in a starting state $\bar{\sigma}$, a sender first assembles a *transaction* tx for the input contract C, where tx indicates f, the sender address sender$[tx]$, and the arguments to f. Let Σ be the CRS generated by the Setup algorithm of the zk-SNARG system (see Def. 4), and pk a table mapping accounts to their public keys. The sender runs $T_{\text{TX}}(C, \bar{\sigma}, tx, sk_{\text{sender}[tx]}, \Sigma, \text{pk})$ as shown in Alg. 2 to create a transaction $\bar{tx}$ for $\bar{C}$.

E. Privacy of ZeeStar

1) Privacy Definition.

Attacker Model. We consider an active PPT adversary statically corrupting a set of dishonest accounts $\mathcal{A}$. The adversary can observe all transactions in the system, and send arbitrary transactions in the name of accounts in $\mathcal{A}$.

Observable Information. To formalize the information the attacker is expected to learn, we define the *observable ideal-world trace* to include the values of all (sub)expressions which are public or owned by dishonest accounts $\mathcal{A}$. Here, values owned by honest accounts $b \notin \mathcal{A}$ are hidden from the adversary and replaced by a placeholder □. In the following, let $\text{obs}_{\mathcal{A}}(C, \sigma, tx)$ denote the ideal-world trace observable by the parties in $\mathcal{A}$ when a transaction tx is executed on contract C in the ideal-world state σ. In Fig. 12, we visualize the observable trace of an example expression for $\mathcal{A} = \{a\}$.

Algorithm 2 Transforming Transactions

1: **procedure** $T_{\text{TX}}(C, \bar{\sigma}, tx, sk_{me}, \Sigma, \text{pk})$
2: Initialize transaction $\bar{tx}$ for same function and sender as tx
3: Copy values of public arguments from tx to $\bar{tx}$.
4: **for** each private argument with value v in tx **do**
5: Add freshly encrypted argument $\text{Enc}(v, \text{pk}(me), r)$ to $\bar{tx}$
6: **for** each argument `arg` introduced by Alg. 1 in C **do**
7: Compute the value x of `arg` according to the rules in Fig. 6, using sk_{me} to decrypt self-owned fields in $\bar{\sigma}$ and public keys in pk to encrypt values for other parties
8: Add x to $\bar{tx}$
9: Use Σ to generate the NIZK proof π and add π to $\bar{tx}$
10: **return** $\bar{tx}$

In particular, the values of `z` and the multiplication expression are hidden from the adversary because they are owned by `bob`. For a formal definition of $\text{obs}_{\mathcal{A}}$, we refer to [9, §4 and §6.2].

Real World. In Alg. 3, we define two algorithms. The left (highlighted) parts define $\text{Real}^{\mathcal{E}}_{\mathcal{A}}$, which models the execution of a sequence of n transactions $tx_{1:n}$ on contract C in the real world, assuming an idealized blockchain with perfect authentication of parties and transactions, and sequential consistency. This protocol uses two sub-protocols $\mathcal{P}$ and $\mathcal{E}$.

The PPT protocol $\mathcal{P}$ captures the behavior of honest accounts. $\mathcal{P}$.Init (Line 6) registers global configuration and keys, and $\mathcal{P}.\text{Tx}(C, \bar{\sigma}_{i-1}, tx_i)$ (Line 13) transforms the transaction tx_i by calling $T_{\text{TX}}(C, \bar{\sigma}_{i-1}, tx_i, sk_{\text{sender}[tx_i]}, \Sigma, \text{pk})$ (Alg. 2) using the information previously received via $\mathcal{P}$.Init.

The PPT protocol $\mathcal{E}$ models the active adversary, which gets access to the secret keys of dishonest accounts $\alpha \in \mathcal{A}$ (Line 7), can craft arbitrary dishonest-sender transactions (Line 10), and observes all transactions on the blockchain (Line 15). $\mathcal{E}$.Decide (Line 16) returns a value specifying whether $\mathcal{E}$ believes to interact with the real world, or a simulated world (see next).

Simulated World. The right (highlighted) parts in Alg. 3 define $\text{Sim}^{\mathcal{E},\mathcal{S}}_{\mathcal{A}}$. This algorithm differs from $\text{Real}^{\mathcal{E}}_{\mathcal{A}}$ in two aspects. First, $\text{Sim}^{\mathcal{E},\mathcal{S}}_{\mathcal{A}}$ additionally keeps track of the ideal-world states σ_i equivalent to the real-world states $\bar{\sigma}_i$ (see Line 5 and Line 14). Here, σ_i is updated according to ideal-world transactions tx_i. For transactions $\bar{tx}_i$ created by $\mathcal{E}$, tx_i is constructed from $\bar{tx}_i$ (see Line 11) as follows. If the proof in $\bar{tx}_i$ is invalid (determined by $\bar{C}$, $\bar{\sigma}_{i-1}$ and $\{\Sigma_\phi\}$), an invalid $tx_i \leftarrow \perp$ is returned. Otherwise, the secret keys of the adversary are used to decrypt self-owned function arguments in $\bar{tx}_i$ and obtain an equivalent ideal-world transaction tx_i.

Second, the steps of honest accounts $\mathcal{P}$ are simulated by a PPT protocol $\mathcal{S}$, which does not get access to private information of honest parties. In particular, $\mathcal{S}$ does not get access to any secret keys (Line 6), and it obtains only observable ideal world traces t of transactions tx_i (Line 13).

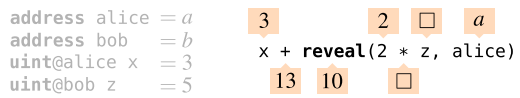


Fig. 12: Observable trace $\text{obs}_{\{a\}}$ of an example expression.

Algorithm 3 $\text{Real}^{\mathcal{E}}_{\mathcal{A}}(C, tx_{1:n})$ and $\text{Sim}^{\mathcal{E},\mathcal{S}}_{\mathcal{A}}(C, tx_{1:n})$

1: Run ZeeStar to transform C to $\bar{C}$
2: For every proof circuit ϕ in $\bar{C}$: $(\Sigma_\phi, \tau_\phi) \leftarrow \text{Setup}(\phi)$
3: For every account α, generate a fresh key pair (pk_α, sk_α)
4: Collect all public keys in the mapping $\text{pk}(\alpha) \mapsto pk_\alpha$
5: Create the initial empty state $\bar{\sigma}_0$ for $\bar{C}$ and σ_0 for C
6: $\mathcal{P}.\text{Init}(C, \text{pk}, \{sk_\alpha\}_{\alpha \notin \mathcal{A}}, \{\Sigma_\phi\})$ $\mathcal{S}.\text{Init}(C, \text{pk}, \mathcal{A}, \{\Sigma_\phi\}, \{\tau_\phi\})$
7: $\mathcal{E}.\text{Init}(C, \text{pk}, \{sk_\alpha\}_{\alpha \in \mathcal{A}}, \{\Sigma_\phi\})$
8: **for** $i = 1, \ldots, n$ **do**:
9: **if** sender$[tx_i] \in \mathcal{A}$ **then**
10: $\bar{tx}_i \leftarrow \mathcal{E}.\text{Tx}(C, \bar{\sigma}_{i-1})$
11: $tx_i \leftarrow \text{GetIdeal}(\bar{tx}_i, \{\Sigma_\phi\}, \bar{C}, \bar{\sigma}_{i-1}, \{sk_\alpha\}_{\alpha \in \mathcal{A}})$
12: **else**
13: $\bar{tx}_i \leftarrow \mathcal{P}.\text{Tx}(C, \bar{\sigma}_{i-1}, tx_i)$ $t \leftarrow \text{obs}_{\mathcal{A}}(C, \sigma_{i-1}, tx_i)$ $\bar{tx}_i \leftarrow \mathcal{S}.\text{Tx}(C, \bar{\sigma}_{i-1}, t)$
14: Run $\bar{tx}_i$ on $\bar{C}$, $\bar{\sigma}_{i-1}$ to get $\bar{\sigma}_i$ and tx_i on C, σ_{i-1} to get σ_i
15: $\mathcal{E}.\text{Notify}(\bar{tx}_i)$
16: **return** $\mathcal{E}.\text{Decide}()$

Privacy. If $\mathcal{S}$ can be instantiated such that any adversary $\mathcal{E}$ cannot distinguish whether it is interacting with real honest parties (in $\text{Real}^{\mathcal{E}}_{\mathcal{A}}$) or with simulated parties (in $\text{Sim}^{\mathcal{E},\mathcal{S}}_{\mathcal{A}}$), then the system respects privacy. This is formalized in Thm. 4.

Theorem 4 (Privacy)**.** *Assume ZeeStar is instantiated with a randomizable (Def. 3) and IND-CPA (Def. 2) encryption scheme, and a zk-SNARG (Def. 4). Let C be 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 $\mathcal{S}^\star$ such that for any PPT adversaries $\mathcal{E}, \mathcal{E}'$, the following advantage is negligible:*

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

2) Proof by Hybrid Argument.

Proof. Let C, $\mathcal{A}$, and $tx_{1:n}$ as in Thm. 4. We next construct PPT simulators $\mathcal{S}_i$ for $i \in \{0, \ldots, 8\}$, following the ideas of the symbolic proof in [9]. By defining $\mathcal{S}^\star := \mathcal{S}_8$, Thm. 4 follows from Lems. 1–9 below and the triangle inequality. □

The Simulator $\mathcal{S}_0$. We define $\text{Sim+}^{\mathcal{E},\mathcal{S}}_{\mathcal{A}}$ equal to $\text{Sim}^{\mathcal{E},\mathcal{S}}_{\mathcal{A}}$, but where $\mathcal{S}$.Init is additionally passed $\{sk_\alpha\}_{\alpha \notin \mathcal{A}}$ in Line 6, and $\mathcal{S}$.Tx is additionally passed tx_i in Line 13. Then, we define $\mathcal{S}_0$ running $\mathcal{P}$ as a sub-protocol as follows. $\mathcal{S}_0.\text{Init}(C, \text{pk}, \mathcal{A}, \{\Sigma_\phi\}, \{\tau_\phi\}, \{sk_\alpha\}_{\alpha \notin \mathcal{A}})$ remembers $\{\tau_\phi\}, \mathcal{A}$ and forwards the other arguments to $\mathcal{P}$.init. $\mathcal{S}_0.\text{Tx}(C, \bar{\sigma}, t, tx)$ simply calls $\mathcal{P}.\text{Tx}(C, \bar{\sigma}, tx)$, ignoring t.

Lemma 1. For any PPT adversaries $\mathcal{E}, \mathcal{E}'$, the advantage $\text{Adv}^{\mathcal{E}'}(\text{Real}^{\mathcal{E}}_{\mathcal{A}}(C, tx_{1:n}),\ \text{Sim+}^{\mathcal{E},\mathcal{S}_0}_{\mathcal{A}}(C, tx_{1:n})) = 0$.

Proof. Straightforward, by construction. □

The Simulator $\mathcal{S}_1$. $\mathcal{S}_1$ is the same as $\mathcal{S}_0$, but with the following modification. Instead of executing Lines 2–3 in Alg. 2 (as part of $\mathcal{P}$.Tx), $\mathcal{S}_1$ reads the function, sender address, and values of the public arguments from the observable trace t. This is possible as these items are public and therefore available in t.

Lemma 2. For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ it is: $\text{Adv}^{\mathcal{E}'}(\text{Sim+}^{\mathcal{E},\mathcal{S}_0}_{\mathcal{A}}(C, tx_{1:n}), \text{Sim+}^{\mathcal{E},\mathcal{S}_1}_{\mathcal{A}}(C, tx_{1:n})) = 0$.

Proof. By construction, both output the same distribution. □

The Simulator $\mathcal{S}_2$. $\mathcal{S}_2$ is the same as $\mathcal{S}_1$, but we change the behavior of Line 7 in Alg. 2 as follows.

Whenever $\mathcal{S}_1$ creates an encryption $Enc_\alpha(T_{\text{plain}}(e))$ for a *dishonest* party $\alpha \in \mathcal{A}$ due to the first case in rule (19), $\mathcal{S}_2$ does not call T_{plain}, but reads the plaintext value v of e from the ideal-world trace t. As e is public, v occurs in t.

Similarly, whenever $\mathcal{S}_1$ creates an encryption $Enc_\alpha(T_{\text{plain}}(e))$ for $\alpha \in \mathcal{A}$ due to Eq. (20), $\mathcal{S}_2$ reads the plaintext value v of e from the ideal-world trace t. As e is revealed to $\alpha \in \mathcal{A}$, v is visible in t.

Also, whenever $\mathcal{S}_1$ calls T_{plain} due to the first case in Fig. 6, $\mathcal{S}_2$ reads the plaintext value v of e from the ideal-world trace t. As e is revealed to the public, v occurs in t.

Finally, when processing the rule (22), $\mathcal{S}_2$ reads the plaintext value v of e_1 from t. Again, as e_1 is public, v occurs in t.

Lemma 3. For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ the advantage $\text{Adv}^{\mathcal{E}'}(\text{Sim+}^{\mathcal{E},\mathcal{S}_1}_{\mathcal{A}}(C, tx_{1:n}), \text{Sim+}^{\mathcal{E},\mathcal{S}_2}_{\mathcal{A}}(C, tx_{1:n}))$ is negligible.

Proof. By construction, the simulators $\mathcal{S}_1$ and $\mathcal{S}_2$ output the same distribution if the values v accessed as described above are correct. By Thm. 3 (correctness) and because n is polynomial, this is the case with overwhelming probability. □

The Simulator $\mathcal{S}_3$. $\mathcal{S}_3$ is the same as $\mathcal{S}_2$, but we modify Eq. (23) as follows. If $\alpha \in \mathcal{A}$, then the value v_1 of e_1 is revealed to the adversary (due to the **`reveal`** expression) and hence available in the trace t. In this case, instead of calling T_{plain} in $\mathcal{S}_2$, $\mathcal{S}_3$ reads v_1 from t.

Lemma 4. For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ the advantage $\text{Adv}^{\mathcal{E}'}(\text{Sim+}^{\mathcal{E},\mathcal{S}_2}_{\mathcal{A}}(C, tx_{1:n}), \text{Sim+}^{\mathcal{E},\mathcal{S}_3}_{\mathcal{A}}(C, tx_{1:n}))$ is negligible.

Proof. By construction, the simulators $\mathcal{S}_2$ and $\mathcal{S}_3$ output the same distribution if the values v_1 accessed as described above are correct. By Thm. 3 (correctness) and because n is polynomial, this is the case with overwhelming probability. □

The Simulator $\mathcal{S}_4$. $\mathcal{S}_4$ is the same as $\mathcal{S}_3$, but instead of generating real proofs in Line 9 of Alg. 2, $\mathcal{S}_4$ uses SimProof (Def. 4) to generate simulated proofs from Σ_ϕ and τ_ϕ.

Lemma 5. For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ it is: $\text{Adv}^{\mathcal{E}'}(\text{Sim+}^{\mathcal{E},\mathcal{S}_3}_{\mathcal{A}}(C, tx_{1:n}), \text{Sim+}^{\mathcal{E},\mathcal{S}_4}_{\mathcal{A}}(C, tx_{1:n})) = 0$.

Proof. Follows from the perfect zero-knowledge property (Def. 4) of the zk-SNARG. □

The Simulator $\mathcal{S}_5$. $\mathcal{S}_5$ is the same as $\mathcal{S}_4$, but instead of encrypting v in Line 5 of Alg. 2, $\mathcal{S}_5$ encrypts the constant 0.

Lemma 6. For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ the advantage $\text{Adv}^{\mathcal{E}'}(\text{Sim+}^{\mathcal{E},\mathcal{S}_4}_{\mathcal{A}}(C, tx_{1:n}), \text{Sim+}^{\mathcal{E},\mathcal{S}_5}_{\mathcal{A}}(C, tx_{1:n}))$ is negligible.

Proof. Follows from the IND-CPA property (Def. 2) of the encryption scheme ($\mathcal{E}$ does not learn the secret key sk_{me}

of the sender), and the fact that n is polynomial. Note that the introduced encryptions of 0 are never decrypted in $\text{Sim+}_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_5}(C, tx_{1:n})$. Further, by the IND-CPA property, SimProof in $\mathcal{S}_4$ and $\mathcal{S}_5$ return indistinguishable proofs, even though its public input changes. □

The Simulator $\mathcal{S}_6$. $\mathcal{S}_6$ is the same as $\mathcal{S}_5$, but we change the behavior of Line 7 in Alg. 2 as follows.

First, the call to T_{plain} in the second case of Fig. 6, is replaced by the constant 0. That is, $\mathcal{S}_6$ simply computes $\text{Enc}(0, pk_{me}, r_i)$. Second, all calls to Enc_α in Fig. 8 for *honest* parties $\alpha \notin \mathcal{A}$ are replaced by fresh encryptions $\text{Enc}(0, pk_\alpha, r)$ of the constant 0.

Lemma 7. For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ the advantage $\text{Adv}^{\mathcal{E}'}(\text{Sim+}_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_5}(C, tx_{1:n}), \text{Sim+}_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_6}(C, tx_{1:n}))$ is negligible.

Proof. Follows from the IND-CPA property (Def. 2) of the encryption scheme ($\mathcal{E}$ does not learn sk_{me} or sk_α for any $\alpha \notin \mathcal{A}$), and the fact that n is polynomial. Again, the introduced encryptions of 0 are never decrypted in $\text{Sim+}_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_6}(C, tx_{1:n})$. □

The Simulator $\mathcal{S}_7$. $\mathcal{S}_7$ is the same as $\mathcal{S}_6$, but we modify Eq. (23) as follows: If $\alpha \notin \mathcal{A}$, $\mathcal{S}_7$ computes $\text{Enc}(0, pk_\alpha, r)$.

Lemma 8. For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ the advantage $\text{Adv}^{\mathcal{E}'}(\text{Sim+}_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_6}(C, tx_{1:n}), \text{Sim+}_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_7}(C, tx_{1:n}))$ is negligible.

Proof. By the randomizability of the encryption scheme (Def. 3), the simulators $\mathcal{S}_7$ and $\mathcal{S}_6$ output a perfectly indistinguishable distribution. □

The Simulator $\mathcal{S}_8$. We finally define, for empty value $\perp$,

$$\mathcal{S}_8.\text{Init}(C, \ldots, \{\tau_\phi\}) := \mathcal{S}_7.\text{Init}(C, \ldots, \{\tau_\phi\}, \perp)$$
$$\mathcal{S}_8.\text{Tx}(C, \bar{\sigma}, t) := \mathcal{S}_7.\text{Tx}(C, \bar{\sigma}, t, \perp).$$

Lemma 9. For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ it is: $\text{Adv}^{\mathcal{E}'}(\text{Sim+}_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_7}(C, tx_{1:n}), \text{Sim}_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_8}(C, tx_{1:n})) = 0$.

Proof. All calls to T_{plain} have been removed in $\mathcal{S}_7$, making rule (15) unreachable. Hence, $\mathcal{S}_7$ no longer accesses sk_{me}. Also, $\mathcal{S}_7$ no longer accesses $tx_{1:n}$. Therefore, the simulators $\mathcal{S}_7$ and $\mathcal{S}_8$ output the same distribution. □

F. Code of zether-confidential

```
pragma zeestar ^1.0.0;

contract ZetherConfidential {
  uint32 MAX = 4294967295;
  uint EPOCH_SIZE = 1;

  uint total;
  mapping(address => uint) lastrollover;
  mapping(address!x => uint32@x<+>) balance;
  mapping(address!x => uint32@x<+>) pending;

  constructor() public {
  }

  function fund() public payable {
    rollover(me);
    require(total + msg.value <= MAX);
    balance[me] = balance[me] + uint32(msg.value);
    total = total + msg.value;
  }

  function transfer(address to, uint32@me<+> val) public {
    rollover(me);
    rollover(to);
    require(reveal(val <= balance[me], all));
    balance[me] = balance[me] - val;
    pending[to] = pending[to] + reveal(val, to);
  }

  function burn(uint32 val) public {
    rollover(me);
    require(reveal(val <= balance[me], all));
    balance[me] = balance[me] - val;
    msg.sender.transfer(val);
    total = total - val;
  }

  function rollover(address y) internal {
    uint e = block.number / EPOCH_SIZE;
    if (lastrollover[y] < e) {
      balance[y] = balance[y] + pending[y];
      pending[y] = 0;
      lastrollover[y] = e;
    }
  }
}
```