zkay: Specifying and Enforcing Data Privacy in Smart Contracts

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

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

Mario Gersbach
ETH Zurich, Switzerland
gmario@student.ethz.ch

Noa Melchior
ETH Zurich, Switzerland
noame@student.ethz.ch

Petar Tsankov
ETH Zurich, Switzerland
petar.tsankov@inf.ethz.ch

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

ABSTRACT

Privacy concerns of smart contracts are a major roadblock preventing their wider adoption. A promising approach to protect private data is hiding it with cryptographic primitives and then enforcing correctness of state updates by Non-Interactive Zero-Knowledge (NIZK) proofs. Unfortunately, NIZK statements are less expressive than smart contracts, forcing developers to keep some functionality in the contract. This results in scattered logic, split across contract code and NIZK statements, with unclear privacy guarantees.

To address these problems, we present the zkay language, which introduces privacy types defining owners of private values. zkay contracts are statically type checked to (i) ensure they are realizable using NIZK proofs and (ii) prevent unintended information leaks. Moreover, the logic of zkay contracts is easy to follow by just ignoring privacy types. To enforce zkay contracts, we automatically transform them into contracts equivalent in terms of privacy and functionality, yet executable on public blockchains.

We evaluated our approach on a proof-of-concept implementation generating Solidity contracts and implemented 10 interesting example contracts in zkay. Our results indicate that zkay is practical: On-chain cost for executing the transformed contracts is around 1M gas per transaction (~0.50US$) and off-chain cost is moderate.

CCS CONCEPTS

• **Security and privacy** → *Cryptography*; *Privacy-preserving protocols*; • **Social and professional topics** → *Privacy policies*; • **Software and its engineering** → *Language features.*

KEYWORDS

Privacy, blockchain, programming language, zero-knowledge proofs

ACM Reference Format:
Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. 2019. zkay: Specifying and Enforcing Data Privacy in Smart Contracts. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3363222

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '19, November 11–15, 2019, London, United Kingdom
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3363222

1 INTRODUCTION

Smart contracts have gained significant popularity in recent years. In a nutshell, they are programs deployed on top of blockchains, such as Ethereum [53], that enable trusted execution without a trusted third party. To benefit from trusted execution with no intermediary, many real-world processes (*e.g.,* trading [1] or insurance [45, 50]) are being ported to smart contracts.

When implementing applications in smart contracts, a major concern is *data privacy*: Smart contract transactions (*i.e.,* function calls initiated by users) are processed by the blockchain's nodes (called miners), which requires the transaction's operations and data to be made available to all nodes. This is problematic for applications that handle sensitive data such as voting schemes [46], the collection of medical data [47], or power consumption measurements [41].

Approaches to Data Privacy in Smart Contracts. One approach to address data privacy is to design new blockchain infrastructures supporting private data. While several such platforms have been proposed [18, 34, 35], they all trade privacy for additional trust assumptions. For example, Hawk [35] and Arbitrum [34] rely on trusted managers and Ekiden [18] leverages trusted hardware; we present details on these trust assumptions in §10.

An alternative approach is leveraging cryptographic primitives to both protect private data and enforce the correctness of computations on blockchains like Ethereum—without undermining their trust model. In particular, Non-Interactive Zero-Knowledge (NIZK) proofs [12, 26] allow users to prove statements about private data without leaking any information besides these statements' truth. Practical NIZK proof constructions have been proposed [6, 11, 27, 42] and made available in Ethereum [16, 23].

The Promise of NIZK Proofs. Intuitively, NIZK proofs enable data privacy by the following construction [15]: First, users encrypt (or hash) their private data and store the resulting ciphertext on the blockchain. Then, to execute a function f of a smart contract modifying private data, the user provides the updated ciphertext (*i.e.,* the ciphertext obtained by encrypting the result of running f on the plaintext private data) along with a NIZK proof certifying that the encrypted values are correct with respect to f.

Limitations of NIZK Proofs. While this construction seems relatively simple, instantiating it for real-world smart contracts is non-trivial due to the following four fundamental challenges C1–4:

(C1) Incompleteness of NIZK Proofs: Real-world smart contracts are implemented in high-level expressive languages (*e.g.,* Solidity [24])

supporting features—such as unbounded state and loops[1]—that cannot be captured by NIZK statements. This is because existing proof constructions reduce the proven statement to an arithmetic circuit that cannot encode arbitrary functions or handle statements of non-constant size. Because of this limitation, developers cannot simply encode the entire function f in a NIZK proof but are forced to use a hybrid solution, where private operations are proven correct using NIZK proofs, but some public operations remain on-chain.

(C2) Knowledge Restrictions: Smart contracts have multiple users with dedicated secrets (*e.g.,* Alice does not know Bob's secrets). However, to invoke f, Alice must have access to *all* private data used by f, otherwise she cannot produce a NIZK proof certifying correctness. For example, Alice cannot increment a counter private to Bob without knowing Bob's secret key and the counter's value.

NIZK Proofs Obfuscate Contracts. Especially due to C1, contracts incorporating NIZK proofs are hard to understand, resulting in two additional challenges. We show a typical example in Fig. 1b (contract) and Fig. 1c (NIZK proof statement); its logic is hard to follow and implementation mistakes are easy to make.

(C3) Obfuscated Logic: Smart contracts incorporating NIZK proofs are obfuscated by logic scattered across off-chain and on-chain computation, making it difficult to determine the intended behavior. As a consequence, unaided development of such contracts is highly error-prone, and the resulting contracts are not easily interpretable.

(C4) Obfuscated Leaks: Because NIZK proofs leak the validity of statements about private values, they *do* leak information. For example, Alice may encrypt a sum of secret values for Bob, proving the ciphertext indeed holds the sum. Thus, we must distinguish intended from unintended information leaks. However, unintended leaks cannot be easily detected in the (already obfuscated) deployed contract, because developers are not forced to make them explicit.

This Work. To address these challenges, we introduce zkay [zi**:** keɪ], a language cleanly separating the task of (i) *specifying* logic and ownership of private data from the task of (ii) *realizing* this specification using NIZK proofs.

Specification. We show an example zkay contract in Fig. 1a. To address the first task, zkay is carefully designed to support fine-grained, expressive, and intuitive privacy specifications allowing developers to specify data ownership by annotating variables as private to particular accounts. Further, it features declassification statements to force developers to explicitly specify what information is revealed by the smart contract. We formally define the data privacy semantics of zkay contracts to cleanly capture the notion of privacy specified by developers.

Addressing the aforementioned challenges, zkay's type system incorporates *privacy types* to statically enforce important properties. First, the type system disallows unrealizable programs (C1), ensuring that a well-typed zkay contract can be realized using NIZK proofs. For example, it ensures that private operations of a function only depend on a constant amount of private data. Second, it restricts operations to be purely based on data available to the caller (C2), ensuring contract functions can always be executed. Third, the logic of zkay programs is easy to follow by just ignoring privacy types (C3). Finally, zkay prevents implicit information leaks, *e.g.,* by disallowing writes of private data to public storage without explicit declassification (C4).

Realization. To realize zkay contracts, we present a fully automated transformation of zkay contracts to equivalent fully public contracts deployable on public blockchains such as Ethereum. The transformed contracts leverage encryption for privacy and NIZK proofs for correctness. Since not all operations can be done privately in proof statements (C1), our transformation produces hybrid contracts performing some public operations on-chain.

As our transformation is provably correct, users can directly work with the original zkay contract to understand its logic, instead of reading the obscure, transformed contract (C3). Further, assuming a Dolev-Yao-style model of NIZK proof and encryption security, transformed contracts are *provably private, i.e.,* equivalent to the original zkay contracts where information leaks are explicit (C4).

System and Experiments. We instantiate our approach by implementing a proof-of-concept system type-checking zkay contracts and transforming them to Solidity contracts [24] executable on Ethereum. We evaluate our approach on 10 example zkay contracts covering a variety of domains. Our results indicate that (i) zkay can express interesting real-world contracts, (ii) programming in zkay offers significant advantages over using NIZK proofs directly, and that (iii) on-chain and off-chain costs of using our transformed contracts are moderate (on-chain costs are roughly 10^6 gas per transaction, corresponding to around 0.50 US$ at time of writing).

Main Contributions. To summarize, our main contributions are:

- A language (§3), called zkay, for writing smart contracts using private data and its formal semantics (§4).
- A privacy-preserving transformation of zkay contracts into equivalent, fully public zkay contracts (§5).
- A formal definition of privacy for zkay contracts along with a proof that each transformed contract is private with respect to its specification contract (§6).
- An implementation and evaluation of our approach instantiated for transforming zkay to Solidity contracts (§8).

Finally, we remark that zkay is not limited to our realization approach, but is also compatible with homomorphic encryption, trusted hardware, or NIZK proofs on hashed states (§9).

2 OVERVIEW

In this section, we use a motivating example to illustrate how one leverages zkay to specify privacy constraints and how these specifications are transformed into a contract using NIZK proofs for enforcement. For readers unfamiliar with blockchains or NIZK proofs, we summarize terminology important for this paper in App. A.

Example: Medical Statistics. Contract `MedStats` in Fig. 1a allows a hospital to collect statistics on blood donors. To this end, the hospital can record every donor's information using function `record`. As arguments, the hospital passes the donor's address and whether that person belongs to a risk group, *e.g.,* due to travel history or a recent illness. Ignoring the blue privacy annotations for now, `record` (i) records the provided data in the `risk` mapping under the donor's key and (ii) increments `count` by one iff the donor belongs

[1]Relying on Ethereum's block gas limit is not straightforward as this limit is dynamic and the amount of gas required may vary with each loop iteration.

```
1  contract MedStats {
2    final address hospital;
3    uint@hospital count;
4    mapping(address!x => bool@x) risk;
5
6    constructor() {
7      hospital = me;
8      count = 0;
9    }
10
11   function record(address don, bool@me r) {
12     require(hospital == me);
13     risk[don] = reveal(r, don);
14     count = count + (r ? 1 : 0);
15   }
16
17   function check(bool@me r) {
18     require(reveal(r == risk[me], all));
19 } }
```

(a) Specification contract `MedStats` in zkay (privacy annotations and reclassifications in blue).

```
1  contract MedStats_bar {
2    final address hospital;
3    bin count;
4    mapping(address => bin) risk;
5
6    constructor(bin v0, bin proof) {
7      hospital = me;
8      count = v0;
9      verify_χ(proof, v0, pk(me));
10   }
11
12   function record(address don, bin r, bin v0, bin v1, bin proof) {
13     require(hospital == me);
14     risk[don] = v0;
15     v2 = count;
16     count = v1;
17     verify_φ(proof, r, v0, v1, v2, pk(don), pk(me));
18   }
19
20   function check(bin r, bool v0, bin proof) {
21     require(v0); verify_ψ(proof, r, risk[me], v0);
22 } }
```

(b) Transformed fully public contract $\overline{\texttt{MedStats}}$ in zkay.

```
1  φ(r, v0, v1, v2, pk_don, pk_me,
2      sk, R0, R1 ) {
3    r_dec = dec_uint(r, sk);
4    v0 == enc(r_dec, R0, pk_don);
5    count_dec = dec_uint(v2, sk);
6    v1_dec = count_dec + (r_dec? 1 : 0);
7    v1 == enc(v1_dec, R1, pk_me);
8  }
```

(c) Proof circuit ϕ for function `record`.

Figure 1: Example illustrating the transformation of a zkay contract into an equivalent but fully public zkay contract.

to a risk group. To check the integrity of the collected statistics, the donor can use the `check` function, which requires that `risk[me]` stores the correct value `r`. Here, **`me`** refers to the caller (analogous to `msg.sender` in Solidity). Observing many successful calls of `check` by other donors, a donor can be confident that the statistics are computed correctly. After recording the statistics of many donors, the hospital may reveal the `count`, possibly protecting it, *e.g.,* by a differential privacy mechanism (not shown).

2.1 Privacy Specification

Whether a donor belongs to a risk group is sensitive information, which can be protected using zkay's type system as we show next.

Specifying Privacy. To protect the information about risk group membership, `MedStats` specifies privacy constraints using *privacy annotations*, enforcing that a value of type $\tau@\alpha$ (consisting of data type τ and privacy type α) can only be read by its *owner* α. For example, Line 3 specifies that `count` is private to its owner `hospital`, meaning that only the hospital may read `count`. We note that, in contrast to reads, writes are not restricted, and therefore anyone may write to `count`. In contrast to `count`, `hospital` in Line 2 has no privacy annotation, meaning that its value is *public* (*i.e.,* any account may read it). To emphasize that a type is public, we may annotate it explicitly as **`@all`**.

For mappings, zkay supports fine-grained privacy specifications where the owner of mapping entries can depend on the mapping key. For example, Line 4 tags the key of mapping `risk` with name `x` and refers to this name in its entry type **`bool`**`@x`. Consequently, `risk[don]` in Line 13 is private to `don`. Explicitly tagging the mapping key is particularly useful for nested mappings. That is, for `m` of type **`mapping(address!x => mapping(address => uint@x))`**, we have that $\texttt{m}[\alpha][\beta]$ is private to α.

In general, a privacy annotation α can be (i) **`me`**, (ii) **`all`**, (iii) a state variable (*i.e.,* a contract field), or (iv) a mapping key tag. For case (iii), zkay's type system ensures the type of α is **`address@all`**, meaning that owners are (publicly known) addresses, and that α is declared **`final`** (*e.g.,* `hospital` in Line 2). zkay's type system ensures that **`final`** variables remain constant. This prevents *ownership transfer*, which we disallow in zkay for simplicity. In §9, we discuss how zkay can be extended to support ownership transfer for fields. For example, modifying `hospital` would implicitly cause `count` to get a new owner. We provide a more formal interpretation of privacy types when we discuss the semantics of zkay (§4).

Type System Exemplified. We now discuss zkay's type system and illustrate how it checks function `record`, addressing challenges C1–C4. We present core rules of zkay's type system (illustrated shortly) in Fig. 2; all rules and further details are given in App. B.

Require: In Line 12, expression `hospital == me` must be public, as the outcome of **`require`** leaks its value (C4, obfuscated leaks).

Reads With Explicit Reclassification: In Line 13, the type system prevents us from directly storing `r` into `risk[don]` to avoid implicitly leaking `r` to the donor (C4, obfuscated leaks). Concretely, the type rule for assignments $L = e$ (Fig. 2) requires (i) typing the target location L as $\tau@\alpha$, (ii) the expression e as $\tau@\alpha'$, and (iii) $\alpha = \alpha' \vee \alpha' = \texttt{all}$. Thus, typing `risk[don]=r` requires instantiating this rule by $L :=$ `risk[don]`, $e :=$ `r`, $\tau :=$ **`bool`**, $\alpha :=$ `don`, and $\alpha' :=$ **`me`**, which violates constraint (iii).

To avoid type errors, we must explicitly reclassify `r` for `don` using **`reveal`**, making the leak explicit (see Line 13). The type rule for **`reveal`** (Fig. 2) only allows reclassifying expressions private to the caller. This is because (i) expressions private to other accounts cannot be read by the caller (C2, knowledge restriction), and (ii) public expressions can be implicitly classified. For example, in Line 8, the constant `0` of type **`uint@all`** is automatically classified

$$\frac{\Gamma \Vdash L\colon \tau@\alpha \quad \Gamma \vdash e\colon \tau@\alpha' \quad (\alpha = \alpha' \vee \alpha' = \texttt{all})}{\Gamma \overset{L=e}{\rightsquigarrow} \Gamma}$$

$$\frac{\Gamma \vdash e\colon \tau@\texttt{me}}{\Gamma \vdash \texttt{reveal}(e, \alpha)\colon \tau@\alpha} \qquad \frac{\Gamma \Vdash L\colon \tau@\alpha \quad \alpha \text{ provably evaluates to caller}}{\Gamma \vdash L\colon \tau@\texttt{me}}$$

Figure 2: Selected typing rules: $\Gamma \vdash e\colon \tau@\alpha$ (resp. $\Gamma \Vdash L\colon \tau@\alpha$) denotes that expression e (resp. location L) is of type $\tau@\alpha$ under the typing context Γ. $\Gamma \overset{P}{\rightsquigarrow} \Gamma'$ denotes that statement P is well-typed and transforms the typing context Γ to Γ'.

for `hospital`. Such classifications can never leak information, but improve the readability of zkay (C3, obfuscated logic).

Reads Without Explicit Reclassification: In order to evaluate the right-hand side in Line 14, the caller must read `count` of type **`uint`**`@hospital` and `r` of type **`bool@me`**. The type system allows reading locations (*i.e.,* variables and mapping entries) only if they are (i) public, or (ii) private to **`me`** (C2, knowledge restriction). In the case of `count`, this is non-obvious as syntactically, `hospital` is not **`me`**. Still, we want to allow Line 14 without forcing an explicit reclassification (C3, obfuscated logic). Thus, zkay employs static analysis (more precisely, Abstract Interpretation [20]) to prove that `hospital` equals **`me`**, which is possible due to the preceding **`require`** statement in Line 12. To reflect this, the type rule for private locations (Fig. 2) requires that α provably evaluates to the caller. Of course, our static analysis is necessarily incomplete, *i.e.,* it may fail to prove that an owner variable equals **`me`**. In this case, a programmer can manually encode additional knowledge using **`require`** statements, thus helping our static analysis.

zkay only enforces privacy type $\alpha \in \{$**`me`**, **`all`**$\}$ for expressions the caller must *read*. In Line 13, the right-hand side is of type **`uint`**`@don`, even though in general, `don != me`. zkay allows this, as the right-hand side is directly assigned, without using it as a sub-expression.

In Line 14, the expression `r ? 1 : 0` has both private (`r`) and public (`0`, `1`) sub-expressions. To prevent implicit information leaks (C4), we type it as **`uint@me`**. Generally, we type native functions (such as `a + b`, `r ? 1 : 0`, etc.) conservatively, making them private to **`me`** if any of their arguments is private to **`me`**.

Loops and Conditionals: For loops and if-then-else statements (not used in `MedStats`), zkay enforces that the condition expression is public, as control flow may implicitly leak the condition's value (C4, obfuscated leaks). Hence, if programmers want control flow to depend on private values, they either need to explicitly declassify these values using **`reveal`**, or re-write the code to make use of conditional expressions e_1`?` e_2 `:` e_3 (where e_1 can be private). Moreover, zkay requires the body and condition of loops to be fully public (*i.e.,* to not involve any private variables). This is necessary since NIZK proof constructions do not support unbounded loops in the statement (C1, incompleteness). In our transformation, we employ a hybrid construction where public loops are executed on-chain.

2.2 Enforcing the Privacy Specification

To enforce privacy specifications in contracts, we provide a transformation from an arbitrary well-typed zkay contract to a semantically equivalent and privacy-preserving yet *fully public* contract (§5). A contract is fully public if all its locations and expressions are public.

Main Ideas. The core ideas of the transformation are (i) to store private values encrypted under the public key of their owner, and (ii) to use NIZK proofs to ensure that state modifications are consistent with the intended operations. In Fig. 1b, we show the transformed version $\overline{\texttt{MedStats}}$ of the contract `MedStats`.

Our transformation ensures that at every execution step in `MedStats`, the transformed contract holds an equivalent state where all private values are encrypted under their owner's public key. For example, assume `risk[0x01]` holds the value **`true`** after Line 13 in `MedStats`. Then, in $\overline{\texttt{MedStats}}$, assuming proof verification (**`verify`**, discussed below) in Line 17 succeeds, after Line 14 `risk[0x01]` holds $\text{Enc}(\texttt{true}, R, \text{Pk}(0x01))$, where R is some randomness and $\text{Pk}(0x01)$ is the public key of the account with address $0x01$.

We now discuss the transformation in more detail on the example of function `record`. When transforming `record`, we first replace the type of its parameter `r` by the ciphertext type **`bin@all`**, as `r` will now hold encrypted values. Since Line 12 (Fig. 1a) is fully public, we do not transform it (cp. Line 13, Fig. 1b).

Ciphertexts, Proofs, and Proof Circuit: To transform Line 13, we must store into `risk[don]` the value of `r`, encrypted for `don`. Because we cannot compute this value on-chain without violating privacy, the caller computes the ciphertext off-chain and provides it as an additional argument `v0`. Then, we store `v0` into `risk[don]` (Line 14, Fig. 1b). To force the caller to provide the correct value for `v0`, we collect a correctness constraint in ϕ (see Fig. 1c, this represents the proof statement verified later). Concretely, Line 4 in ϕ checks that `v0` is the result of encrypting $\texttt{r}_{\texttt{dec}}$ using randomness `R0` and key $\texttt{pk}_{\texttt{don}}$, the public key of `don`. Here, we obtain $\texttt{r}_{\texttt{dec}}$ by decrypting `r` in Line 3 using the caller's secret key `sk`. The highlighted arguments `R0` and `sk` of ϕ (called *secret arguments*) cannot be provided on-chain because knowing `R0` enables guessing attacks on `v0` and knowing `sk` allows decrypting `r`. Upon calling `record`, the caller provides a NIZK proof `proof` certifying she knows these secret values such that ϕ is satisfied together with the remaining arguments provided on-chain (see next). This proof is verified in Line 17 of Fig. 1b, where the arguments of **`verify`** serve as the public arguments of ϕ (**`pk`** fetches public keys). Due to the nature of NIZK proofs, verification does not leak any information about the secret arguments besides their existence. Proof verification $\textbf{verify}_{\phi}$ can be realized on public blockchains using tools like ZoKrates [23].

Finally, we transform Line 14 (Fig. 1a) to Line 16 (Fig. 1b), replacing the private expression `count + (r ? 1 : 0)` by an argument `v1`. Again, ϕ checks the correctness of `v1` (Lines 5–7). Note that to read the original value of `count` in Line 5 of ϕ, we must record it in Line 15 of Fig. 1b, as we overwrite `count` in Line 16.

Hybrid Approach: Our approach is hybrid in the sense that some operations are executed inside the contract, outside the proof circuit. For instance, mapping entries are always resolved on-chain (*e.g.,* `risk[me]` in Line 21 of Fig. 1b) so to avoid passing whole mappings to the verifier (see C1 in §1). This requires us to disallow encrypting whole mappings and force mapping keys to be public.

Transactions: To enable calling the transformed functions, we also transform transactions. For example, we transform a transaction `record(0x01,false)` to `record(0x01,r,v0,v1,`p`)`, where `r`, `v0`, `v1` are computed off-chain in accordance with ϕ (*e.g.,* `v0` is the encryption of **`false`** for `0x01`), and p is an appropriate NIZK proof.

$$L ::= \text{id} \mid L[e] \quad \text{(Location)} \qquad \alpha ::= \texttt{me} \mid \texttt{all} \mid \text{id} \quad \text{(Privacy type)}$$

$$e ::= c \mid \texttt{me} \mid L \mid \texttt{reveal}(e, \alpha) \mid \ominus e \mid e_1 \oplus e_2 \mid e_1 ? e_2 : e_3 \mid \texttt{pk}(e) \mid \cdots \quad \text{(Expression)}$$

$$\tau ::= \texttt{bool} \mid \texttt{uint} \mid \texttt{address} \mid \texttt{bin} \mid \texttt{mapping}(\tau_1 \Rightarrow \tau_2 @ \alpha_2) \mid \texttt{mapping}(\texttt{address!}\,\text{id} \Rightarrow \tau @ \alpha) \quad \text{(Data type)}$$

$$P ::= \texttt{skip} \mid \tau @ \alpha\ \text{id} \mid L = e \mid P_1; P_2 \mid \texttt{require}(e) \mid \texttt{if}\ e\ \{P_1\}\ \texttt{else}\ \{P_2\} \mid \texttt{while}\ e\ \{P\} \mid \texttt{verify}_\phi(e_0, e_1, \ldots, e_n) \quad \text{(Statement)}$$

$$F ::= \texttt{function}\ f(\tau_1 @ \alpha_1\ \text{id}_1, \ldots, \tau_n @ \alpha_n\ \text{id}_n)\ \texttt{returns}\ \tau @ \alpha\ \{P; \texttt{return}\ e;\} \quad \text{(Function)}$$

$$C ::= \texttt{contract}\ \text{id}\{(\texttt{final})?\ \tau_1 @ \alpha_1\ \text{id}_1;\ \ \ldots\ \ (\texttt{final})?\ \tau_n @ \alpha_n\ \text{id}_n;\ \ F_1 \ldots F_m\} \quad \text{(Contract)}$$

Figure 3: Syntax of zkay, where f and 'id' are identifiers, c is a constant, and ϕ an arithmetic circuit. Native functions are highlighted.

Privacy. Note that it is not immediately clear what it means for a contract to be private. Prohibiting *any* leak of information is too restrictive: even the specification contract `MedStats` leaks some information about its private data (*e.g.,* due to the declassification in Line 18), which is reflected also in its transformed variant $\overline{\texttt{MedStats}}$.

In our work, we introduce a formal definition of privacy taking this subtlety into account. Privacy of a contract is always defined with respect to a specification contract. We define contract $\overline{\texttt{MedStats}}$ to be private w.r.t. contract `MedStats` iff any transaction on $\overline{\texttt{MedStats}}$ does not leak more information than the analogous transaction on `MedStats` executed in an ideal world where private values are kept secret for the respective owner. We formalize this notion by introducing *traces* leaked by transactions on zkay contracts. We prove that our transformation respects privacy (§6) by showing that any trace $\bar{t}$ of a transaction in $\overline{\texttt{MedStats}}$ can be simulated from the corresponding trace in `MedStats` by producing a trace indistinguishable from $\bar{t}$.

3 THE ZKAY LANGUAGE

Fig. 3 shows the syntax of zkay. In order to focus on key insights, zkay is deliberately kept simple.

zkay consists of (memory) locations, expressions, data and privacy types, statements, functions, and contracts. A type declaration ($\tau@\alpha$) in zkay consists of a *data type* (τ), and a *privacy type* (α) specifying the owner of a construct. Privacy types consist of **me**, a pseudo-address indicating public accessibility (**all**), and identifiers (covering state variables and mapping key tags). For readability, we often omit **all**, writing τ instead of τ@**all**. Locations (L) consist of contract field identifiers, function arguments and local variables ('id', alphanumeric strings), and mapping entries ($L[e]$).

The only zkay-specific expressions are the runtime address of the caller (**me**), and re-classification of information (**reveal**). The highlighted expressions can be viewed as evaluations of so-called *native functions* $g(e_1, \ldots, e_n)$, including standard arithmetic and boolean operators (captured by $\ominus, \oplus$). The expression **pk**(e) returns the public key of the address expression e from a public key infrastructure. It is straightforward to extend zkay with additional native functions (as indicated by '$\cdots$' in Fig. 3). For simplicity, we don't discuss the handling of calls to functions of the same or other contracts. zkay can, however, support such calls whenever the called function bodies are statically known; we discuss this in §9.

In addition to well-known data types (**bool** and **uint**), zkay supports addresses indicating accounts (**address**), and binary data capturing NIZK proofs, public keys and ciphertexts (**bin**). In addition, types include mappings (**mapping**$(\tau_1 \Rightarrow \tau_2@\alpha_2)$) and *named* mappings (**mapping**$(\texttt{address!}\,\text{id} \Rightarrow \tau@\alpha)$, defining name 'id' for the key of the map to be used in the key type $\tau@\alpha$. It is straightforward to extend zkay with additional types, such as floats, structs, and arrays (conceptually, arrays are equivalent to **mapping**$(\texttt{uint} \Rightarrow \tau@\alpha)$).

Statements (P) in zkay are mostly standard. To declare a local variable, we write $\tau@\alpha$ id. If e does not evaluate to **true**, **require**(e) throws an exception. Finally, zkay supports NIZK proof verification (**verify**). We only include this statement to express transformed contracts and assume specification contracts never use **verify**. In statement $\texttt{verify}_\phi(e_0, e_1, \ldots, e_n)$, the *proof circuit* ϕ is an arithmetic circuit (*i.e.,* a loop-free mathematical function) taking n public and m secret arguments, and returning a number in $\{0, 1\}$. The verification statement verifies that e_0 is a valid NIZK proof certifying there exist m secret values $v_1, \ldots, v_m$ such that ϕ returns 1 when given n public arguments $e_1, \ldots, e_n$ and secret arguments $v_1, \ldots, v_m$. Proof verification does not leak any information about the secret arguments of the circuit ϕ other than the fact that ϕ returns 1. We could easily include cryptocurrency transfers (`transfer` in Solidity) in zkay, but omit them for simplicity.

While we only discuss functions (F) which return a value, zkay also allows constructors and functions without return values (*e.g.,* see Fig. 1a). Contracts (C) consist of contract field declarations and function declarations, where contract fields may be declared **final**.

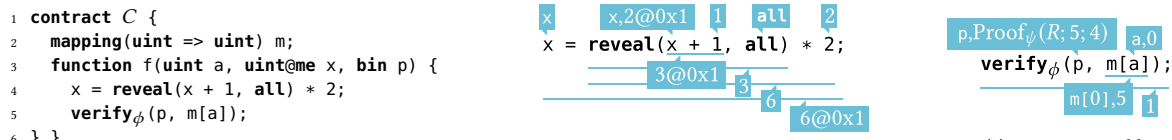
4 SEMANTICS BY EXAMPLE

We now define how transactions update the contract state by evaluating zkay statements, expressions and locations, and how transactions generate *traces* containing information about intermediate execution steps, modeling leaked information.

4.1 Traces

Every execution in zkay (*e.g.,* expression evaluation or statement execution) produces a *trace*. Intuitively, the trace defines which information is leaked during execution (including control flow, reads, writes, and calculations) and to whom it is leaked. Traces are essential to define zkay's privacy notion (§6). Formally, a trace t is a sequence of entries $v_i@a_i$ for values v_i and *privacy levels* a_i. The latter is either (i) an address, indicating that v_i is private to a_i and can only be seen by a_i, or (ii) **all**, indicating that v_i is public. For brevity, we usually omit @**all** from traces.

We write $\text{Tx}^{(a)}_{C.f}(v_{1:n})$ to denote a transaction issued by address a, calling function f of contract C with arguments $v_{1:n}$. We write $\langle T, \sigma\rangle \overset{t}{\Rightarrow} \langle \sigma', v\rangle$ to denote that executing transaction T on state σ (introduced next) produces the trace t, updates the state to σ', and returns the value v. If T throws an exception, we set $v =$ **fail**.



(a) Example contract C.

(b) Trace emitted by Line 4 of C.

(c) Trace emitted by Line 5 of C.

Figure 4: zkay semantics illustrated for an example transaction where address 0x1 calls $\texttt{f}(0, 2, \text{Proof}_\psi(R; 5; 4))$ on C, assuming $\texttt{m[0]} = 5$.

4.2 Example Transaction

We introduce key aspects of zkay's semantics using an example (formal semantics for zkay is provided in App. C). Consider the contract in Fig. 4a consisting of a mapping field `m` and a function `f`. In Line 5, ϕ is an arithmetic circuit defined as follows (v_1 is a public and v_2 a secret argument of ϕ):

$$\phi(v_1, v_2) = 1 \quad \Longleftrightarrow \quad v_1 - 1 = v_2.$$

States and Values. The state σ of a contract specifies the values of all fields. For our example, assuming that `m[0]` holds value 5, we write $\sigma = \{\texttt{m} \mapsto \{0 \mapsto 5\}\}$. We use symbolic representations for values of type **`bin`** (*i.e.,* keys, ciphertexts and NIZK proofs). Specifically, we write $\text{Enc}(v, R, \text{Pk}(a))$ to denote the encryption of v using the public key of a and symbolic randomness R. NIZK proofs are represented by $\text{Proof}_\phi(R; v_{1:n}; v'_{1:m})$, where ϕ is the proof circuit, R is symbolic randomness used to generate the proof, and $v_{1:n} := v_1, \ldots, v_n$ (resp. $v'_{1:m}$) are the public (resp. secret) arguments for ϕ bound in the proof. A similar notation is introduced in [3]. We say that $\text{Proof}_\phi(R; v_{1:n}; v'_{1:m})$ is *valid* iff $\phi(v_{1:n}, v'_{1:m}) = 1$.

Assume a caller with address 0x1 starts a transaction calling `f` with arguments $(0, 2, \text{Proof}_\psi(R; 5; 4))$. Here, it is $\psi = \phi$ (though in general, we may have $\psi \neq \phi$). Before Line 4 (Fig. 4a), the state is:

$$\sigma' = \{\texttt{m} \mapsto \{0 \mapsto 5\}, \texttt{a} \mapsto 0, \texttt{x} \mapsto 2, \texttt{p} \mapsto \text{Proof}_\psi(R; 5; 4), \textbf{me} \mapsto \text{0x1}\}$$

Expressions and Assignments. We now describe how the assignment in Line 4 is evaluated. Fig. 4b illustrates which trace entries $v_i@a_i$ are emitted by each evaluation step.

Evaluation of the right-hand side expression starts at the leafs. Evaluating the constant 1 emits trace entry 1 with privacy level **`all`** (omitted), because constants are public. The location `x` is private to the caller, which has address 0x1. When reading `x`, two trace entries are emitted. First, we emit `x` to indicate the accessed location. This entry is public to model the fact that accessed memory locations cannot be hidden. This is in contrast to the *value* of `x`, which is added to the trace as 2@0x1 with privacy level 0x1. The expression `x + 1` is private to the caller according to the type system, hence its evaluation result 2 + 1 = 3 is added to the trace as 3@0x1. This reflects that nobody except address 0x1 can see this result. The **`reveal`** expression emits **`all`** (evaluating its second argument) and reveals the value 3 of its first argument by emitting the public trace entry 3. Note how the value 3 is now visible to everyone. Multiplying it with the constant 2 emits public trace entries 2,6.

For the left-hand side of the assignment, the location `x` is added to the trace as a public entry. Note how the right-hand side is implicitly classified for 0x1: the public value is written to `x`, which is private to the caller according to the type system. Hence, a final entry 6@0x1 is added to the trace and the value of `x` in σ' is updated to 6 (the value stored in the state has no privacy level attached).

In summary, the trace generated when executing Line 4 is

$$\texttt{x}, 2@\text{0x1}, 1, 3@\text{0x1}, \textbf{all}, 3, 2, 6, \texttt{x}, 6@\text{0x1}$$

and the new state is

$$\{\texttt{m} \mapsto \{0 \mapsto 5\}, \texttt{a} \mapsto 0, \texttt{x} \mapsto 6, \texttt{p} \mapsto \text{Proof}_\psi(R; 5, 4), \textbf{me} \mapsto \text{0x1}\}.$$

Proof Verification. Next, we describe how the proof `p` is verified in Line 5. Fig. 4c illustrates the emitted trace entries.

First, `p` is evaluated to the proof $P := \text{Proof}_\psi(R; 5; 4)$, emitting entries `p`,$P$. Then, `m[a]` is evaluated to 5, which emits entries `a`,0 (from evaluating `a`) followed by `m[0]`,5. Note the entry `m[0]` in the trace: in contrast to the (syntactic) location `m[a]`, this so-called *runtime location* specifies the key value.

Next, the proof P is verified. This includes checking that (i) the circuit bound in P (here: ψ) equals the target circuit of the verification statement (here: ϕ), (ii) the public value bound in P (here: 5) matches the value of the second argument of **`verify`** (here: `m[a]` = 5), and (iii) P is valid (*i.e.,* checking that $\psi(5, 4) = 1$). Because verification is successful, the **`verify`** statement emits a public trace entry 1. Note how no other trace entries are generated by **`verify`**, reflecting the zero-knowledge nature of proof verification.

The transaction is finished successfully. The final contract state only retains the values of contract fields and is hence equal to the initial state σ in our example (the value of `m` was not updated).

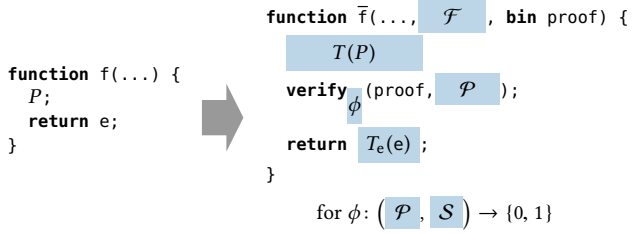
In case of verification failure (*e.g.,* if $\phi \neq \psi$ in Line 5), an exception would be thrown: execution is stopped immediately and the state is rolled back to the state before the transaction. In this case, the trace would still contain the previously collected information, but end with a special entry "rollback".

5 TRANSFORMATION

We now describe how to transform a zkay contract C to a fully public zkay contract $\overline{C}$ (§5.1–§5.3) while preserving privacy (discussed in §6). Because $\overline{C}$ has a different interface than C, we also discuss how to transform transactions T on C to transactions $\overline{T}$ on $\overline{C}$ (§5.4).

Correctness. By construction, our transformation ensures correctness, as formalized in Thm. 1.

Theorem 1 (Correctness). Given a contract C and its transformation $\overline{C}$, for any two *equivalent* states σ and $\overline{\sigma}$ and any transaction $\overline{T}$, running $\overline{T}$ on $\overline{C}$, $\overline{\sigma}$ either throws an exception or there exists a transaction T for the same function and using the same public arguments as $\overline{T}$ such that: $\langle T, \sigma\rangle \overset{t}{\Rightarrow} \langle \sigma', v\rangle$ in C and $\langle \overline{T}, \overline{\sigma}\rangle \overset{\overline{t}}{\Rightarrow} \langle \overline{\sigma}', \overline{v}\rangle$ in $\overline{C}$ for some σ',v *equivalent to* $\overline{\sigma}'$,$\overline{v}$ and some traces t, $\overline{t}$.



(a) Transformation of a zkay function.

$$out(e, \alpha) ::= v_i \qquad \text{(invariant: } e\texttt{@me} \text{ or } e\texttt{@all}\text{)}$$
$$\mathcal{F} \leftarrow \mathcal{F}, v_i$$
$$\mathcal{P} \leftarrow \mathcal{P}, v_i, \texttt{pk}(\alpha)$$
$$\mathcal{S} \leftarrow \mathcal{S}, R_i$$
$$\phi \leftarrow \phi;\ v_i == \texttt{enc}(\ T_\phi(e)\ , R_i, \texttt{pk}(\alpha))\ ;$$

(b) Transformation $out(e, \alpha)$. If $\alpha = \texttt{all}$, the highlighted part is omitted.

$$in(e, \alpha) ::= \texttt{dec}_\tau(\ v_i, \texttt{sk})\ \ \text{(invariant: } e@\alpha,\ \alpha \in \{\texttt{me}, \texttt{all}\}\text{)}$$
$$\text{add to } T(P) \ :\ \tau'\texttt{@all}\ v_i\ =\ T_e(e); \qquad \text{for } \tau' ::= \begin{cases} \tau & \alpha = \texttt{all} \\ \texttt{bin} & \alpha \neq \texttt{all} \end{cases}$$
$$\mathcal{P} \leftarrow \mathcal{P}, v_i$$
$$\mathcal{S} \leftarrow \mathcal{S}, \texttt{sk}$$

(c) Transformation $in(e, \alpha)$. If $\alpha = \texttt{all}$, the highlighted part is omitted.

$$T(P_1; P_2) ::= T(P_1); T(P_2) \tag{1}$$
$$T(L@\alpha = e@\alpha) ::= T_L(L) = T_e(e) \qquad \text{(we use } T_L = T_e\text{)} \tag{2}$$
$$T(L@\alpha = e@\texttt{all}) ::= T_L(L) = out(e, \alpha) \qquad (\alpha \neq \texttt{all}) \tag{3}$$
$$T(\texttt{require}(e)) ::= \texttt{require}(T_e(e)) \tag{4}$$
$$T(\texttt{while}\ e\ \{P\}) ::= \texttt{while}\ e\ \{P\} \qquad (P \text{ is fully public}) \tag{5}$$
$$T(\texttt{if}\ e\ \{P_1\}\ \texttt{else}\ \{P_2\}) ::= \texttt{if}\ T_e(e)\ \{T(P_1, T_e(e))\}\ \texttt{else}\ \{T(P_2, T_e(!e))\} \tag{6}$$

(d) Transforming statements using T.

$$T_e(c) ::= c \qquad \text{const } c \tag{7}$$
$$T_e(\text{id}) ::= \text{id} \qquad \text{var id } (\star) \tag{8}$$
$$T_e(L[e]) ::= T_L(L)[T_e(e)] \qquad \text{mapping entry} \tag{9}$$
$$T_e((e_1 + e_2)@\texttt{all}) ::= T_e(e_1) + T_e(e_2) \qquad \text{native functions} \tag{10}$$
$$T_e(\texttt{reveal}(e, \alpha)) ::= out(e, \alpha) \qquad \text{(invariant: } e\texttt{@me}\text{)} \tag{11}$$
$$T_e(e@\alpha) ::= out(e, \alpha) \qquad \text{(invariant: } e\texttt{@me}\text{)} \tag{12}$$

(e) Transforming expressions in zkay using T_e. For private function arguments id, $\star$ adds an additional correctness constraint to ϕ.

$$T_\phi(c) ::= c \qquad \text{const } c \tag{13}$$
$$T_\phi(L@\alpha) ::= in(L, \alpha) \qquad \text{(invariant: } \alpha \in \{\texttt{all}, \texttt{me}\}\text{)} \tag{14}$$
$$T_\phi(\texttt{reveal}(e, \alpha)) ::= T_\phi(e) \tag{15}$$
$$T_\phi(e_1 + e_2) ::= T_\phi(e_1) + T_\phi(e_2) \qquad \text{native functions} \tag{16}$$

(f) Transforming expressions in the proof circuit using T_ϕ.

Figure 5: Overview of zkay transformations. We write $e@\alpha$ to indicate that e has privacy type α. The symbol v_i denotes a fresh variable.

Formally, value v in C is *equivalent to* value v' in $\overline{C}$, if either v is public and $v = v'$, or v is private to a and $v' = \text{Enc}(v, R, \text{Pk}(a))$ for some randomness R. As a natural extension, state σ in C is equivalent to state σ' in $\overline{C}$ if all values in σ and σ' are equivalent.

5.1 Transformation Overview

The general idea of transforming a contract C to $\overline{C}$ is to (i) replace private expressions by encrypted arguments provided by the caller, (ii) replace declassified expressions by cleartext arguments provided by the caller, and (iii) require the caller to provide NIZK proofs certifying correctness (*i.e.,* equivalence) of these arguments w.r.t. C.

Fig. 5 shows an overview of our transformation. To avoid notational clutter, the figure does not describe how to transform the types of private locations. Because these hold encrypted values in $\overline{C}$, our transformation changes their type to **bin@all**. An example is `count` in Line 3 of Fig. 1a, transformed to Line 3 of Fig. 1b.

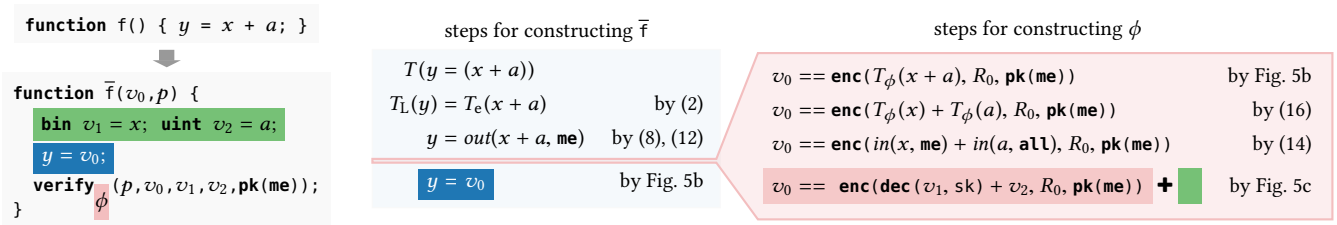
Transforming Functions. A well-typed zkay contract C is transformed by transforming all its functions according to Fig. 5a. The function body and returned expression are transformed using statement transformation T (Fig. 5d) and expression transformation T_e (Fig. 5e), respectively. The function's parameters are extended by parameters $\mathcal{F}$ and a NIZK proof. During transformation of the body and return value, we collect correctness constraints on $\mathcal{F}$ in a proof circuit ϕ. The proof `proof` is verified w.r.t. ϕ, public arguments $\mathcal{P}$ and secret arguments $\mathcal{S}$ (bound to the proof during proof generation) at the end of the body. No verification statement is added if ϕ is empty (*i.e.,* if no correctness constraints were collected).

Encoding Proof Circuits. Up until now, we have viewed proof circuits ϕ as abstract mathematical functions. We will later use the NIZK verifier generation tool ZoKrates [23] to instantiate $\texttt{verify}_\phi$ for a given ϕ, hence the latter ultimately needs to be encoded in ZoKrates' DSL. To avoid introducing this DSL, from now on we encode ϕ using zkay assignments, variable declarations, and boolean expression (*e.g.,* equality) constraints. We augment expressions by asymmetric encryption and decryption with standard semantics: the expression $\texttt{enc}(x, R, k)$ encrypts x using randomness R and public key k to yield the (symbolic) value $\text{Enc}(x, R, k)$ of data type **bin**, while $\texttt{dec}_\tau(x, k)$ decrypts x of data type **bin** using the secret key k and returns a value of data type τ. We define ϕ to return 1 iff evaluating ϕ according to zkay semantics results in all constraints being satisfied. Fig. 1c shows an example of a proof circuit encoding.

5.2 Transformation Example

We now provide an intuition of the transformation defined in Fig. 5 using a concrete example. Particularly, we discuss how the function `f` shown in Fig. 6a is transformed step-by-step (Fig. 6b) to $\overline{\texttt{f}}$.

Transforming Statements. We begin by transforming the body of `f` using the statement transformation T formally defined in Fig. 5d. Fig. 6b (left, first two lines) shows how we apply rule (2).



(a) Transforming `f` to $\overline{f}$.

(b) Step-by-step transformation of the assignment $y@\texttt{me} = x@\texttt{me} + a@\texttt{all}$ (emitted parts are highlighted).

Figure 6: Transforming an example function `f`, where x and y are fields private to `me` and a is a public field. Data types are not shown.

In general, T ensures that intermediate states in C and $\overline{C}$ are equivalent. More precisely, for any statement P, it ensures the following invariant: *Assuming the constraints expressed in ϕ hold at the end of $\overline{f}$, P is equivalent to $T(P)$.* Formally, statement P in C is equivalent to statement P' in $\overline{C}$ if for any states σ in C equivalent to σ' in $\overline{C}$, successfully running P in σ and P' in σ' results in equivalent states. Here, "successfully" means in absence of exceptions.

Transforming Expressions. Next, we transform the right-hand side $x + a$ using expression transformation T_e (Fig. 5e). At a high-level, T_e recursively transforms expressions while (i) leaving public expressions unchanged, and (ii) substituting private expressions by fresh function arguments v_i whose correctness is enforced by adding constraints to ϕ. In our case, the argument $x + a$ of T_e is private to `me` and we hence apply rule (12). Because $x + a$ cannot be evaluated on-chain in $\overline{f}$ without violating privacy, we require the caller to pass the encryption of $x + a$ via a fresh function argument, and to prove its correctness. Conceptually, this amounts to evaluating $x + a$ in ϕ, and making the result available within $\overline{f}$ (*i.e.,* moving $x + a$ out of ϕ). This is achieved using $out(x + a, \texttt{me})$, discussed in the next paragraph.

We apply location transformation T_L to the left-hand side y. Because T_L is equal to T_e, except for $\star$ in Fig. 5e (discussed in §5.3), we do not discuss it further. In our example, we apply rule (8).

In general, for any expression e, expression transformation T_e ensures the following invariant : *Assuming the constraints expressed in ϕ hold at the end of $\overline{f}$, e in C is equivalent to $T_e(e)$ in $\overline{C}$.* Formally, expression e in C is equivalent to e' in $\overline{C}$ if for any states σ in C equivalent to σ' in $\overline{C}$, successfully evaluating e in σ and e' in σ' yields equivalent values. T_L ensures that $T_L(L)$ in $\overline{C}$ evaluates to the same runtime location as L in C.

Moving Values out of the Proof Circuit. The transformation *out* (Fig. 5b) ensures that the correct encryption of $x + a$ is computed in ϕ and "moved out" to $\overline{f}$. It performs two steps: First, it replaces $x + a$ by a fresh function argument v_0 (see also Fig. 6a) and thereby concludes the transformation in $\overline{f}$ (see highlighted in Fig. 6). Then, it continues transformation inside the proof circuit ϕ (see right box in Fig. 6b). It makes v_0 available in ϕ by adding v_0 to the public arguments $\mathcal{P}$, and adds a constraint to ϕ ensuring that v_0 is a proper encryption of the correct cleartext value of $x + a$. This involves adding the encryption key `pk(me)` to $\mathcal{P}$ and fresh randomness R_0 to the secret proof circuit arguments $\mathcal{S}$.[2] R_0 must be secret, as the ciphertext would otherwise be vulnerable to guessing attacks.

Transforming Expressions in the Proof Circuit. To ensure correctness of the computation, we need to compute the cleartext value of $x+a$ in ϕ (note that this remains private), which is achieved using the transformation T_ϕ (Fig. 5f). Addition can directly be performed in ϕ and we apply rule (16). Next, we apply rule (14) to make the cleartext values of variables x and a accessible to ϕ. This is, we "move them into" ϕ using *in* (discussed next).

In general, for any expression e, $T_\phi(e)$ ensures that *successfully evaluating e in C results in the same (unencrypted) value as evaluating $T_\phi(e)$ in ϕ during verification of ϕ in $\overline{C}$.*

Moving Values into the Proof Circuit. Note that because x is private in `f`, it is encrypted in $\overline{f}$. To make the cleartext value of x available in ϕ, *in* (Fig. 5c) performs the following steps. First, it passes the current (encrypted) value of x from $\overline{f}$ to ϕ by (i) adding a new public proof argument v_1 to ϕ, (ii) storing the current state of x in a fresh local variable v_1 in $\overline{f}$ (see highlighted in Fig. 6a, this step is important as the value of x could in general change later), and (iii) passing v_1 to ϕ at `verify`. As a result, v_1 in ϕ will contain the current encrypted value of x. Second, *in* decrypts v_1 using the caller's secret key `sk`, which is added as a private argument to ϕ. The cleartext value of a is similarly made available in ϕ via a public proof circuit argument v_2. Because a is public in `f`, a is not encrypted in $\overline{f}$ and no decryption is required.

The resulting constraint in ϕ (see highlighted in Fig. 6b, right) enforces that v_0 is a proper encryption of $x + a$, according to `f`, under the caller's public key.

Because our transformation invariants ensure that for $in(e, \alpha)$, e is never private to somebody else than the caller, *in* never requires somebody else's private key (cp. C2, knowledge restrictions).

5.3 Additional Rules

We now describe some additional transformation rules.

Statements. Rule (3) handles implicit classifications by moving the appropriately encrypted value out of the proof circuit. Our type system enforces `while` loops to be fully public, meaning that their termination conditions and bodies do not involve private variables. Hence, they are left untransformed by T. We transform `if` e $\{P_1\}$ `else` $\{P_2\}$ by individually transforming e, P_1 and P_2 (note

[2] To simplify notation, we directly refer to `pk(me)` inside ϕ. In our implementation, we actually introduce a fresh parameter to pass the public key (cp. $\texttt{pk}_{\texttt{me}}$ in Fig. 1c)

that e may include declassifications). Since transformation of P_1 (resp. P_2) may add constraints to ϕ which are only relevant if e evaluates to true (resp. false), we extend the functions T and T_e to take a *guard condition* b as an optional second argument. All constraints c added to ϕ by $T(P, b)$ or $T_e(P, b)$ are only enforced if b is true by replacing them by $!b \mid\mid c$. To avoid clutter, we do not incorporate guard conditions in Fig. 5.

Expressions. When being transformed with T_e, private function parameters require adding a correctness constraint (not shown) to ϕ ensuring the argument is indeed a value encrypted for the correct account (see $\star$ in rule (8)). This is not required for T_L.

Rules (10) and (11) together ensure that public expressions are recursively transformed until a declassification is hit. For example, T_e transforms a@`all` + `reveal`(x@`me` + 1, `all`) to $a@ + v_i$, where the function argument v_i containing the revealed cleartext has to be provided by the caller.

5.4 Transforming Transactions

Transforming a transaction T for C to $\overline{T}$ for $\overline{C}$ is simple at a conceptual level: the function arguments for $\overline{T}$ are constructed such that proof verification in $\overline{C}$ succeeds. Function arguments public in T can directly be used in $\overline{T}$, while private function arguments are encrypted under the caller's public key in $\overline{T}$. Additional function arguments v_i introduced by transformation are chosen in accordance with the constraints generated when v_i is introduced, see the update of ϕ in Fig. 5b. We note that transforming T is only allowed if it does not throw an exception on C for the current state.

To generate the `proof`, we must determine both public ($\mathcal{P}$) and secret ($\mathcal{S}$) arguments to ϕ. To determine $\mathcal{P}$, we simulate the partial transaction $\overline{T}$ (which does not yet include a proof) on $\overline{C}$. For $\mathcal{S}$, we simply provide fresh randomness and the caller's secret key.

6 PRIVACY MODEL FOR ZKAY

We now define privacy for zkay contracts and prove that any transformed contract $\overline{C}$ is private with respect to its original contract C. We first define a model of an attacker interacting with $\overline{C}$ in the real world (§6.1), and how C is executed in an ideal world (§6.2). Finally, we prove that an attacker can learn nothing more from transactions on $\overline{C}$ in the real world than on C in the ideal world (§6.3).

6.1 Attacker Model

We consider an active attacker interacting with a public blockchain as modeled by our semantics (§4). We model an attacker as the set $\mathcal{A}$ of addresses she controls (*i.e.,* the attacker knows the secret keys of accounts in $\mathcal{A}$) and call all other accounts *honest*. Let $\overline{C}$ be the result of transforming a contract C. The attacker $\mathcal{A}$ can interact with the fully public contract $\overline{C}$ in the following two ways. First, she can observe all traces of any transaction on $\overline{C}$, including transactions by honest callers. This captures the behavior of public blockchains such as Ethereum, where miners (including the attacker) run contracts locally and thereby learn every intermediate execution step. Second, the attacker can issue transactions on $\overline{C}$ on behalf of any account in $\mathcal{A}$, capturing potentially malicious calls by dishonest accounts.

We adopt a symbolic view in the standard Dolev-Yao model [22], where cryptographic primitives are assumed to be perfect. As usual, we use distinct sets $\mathsf{R}_{\mathsf{adv}}$ and $\mathsf{R}_{\mathsf{hon}}$ for randomness generated by the attacker and, respectively, honest accounts (cp. [3]).

We define the attacker capabilities by which (symbolic) values she can distinguish and which not (*e.g.,* based on cryptographic operations and comparisons). We assume a strong attacker who can distinguish almost all inequal values, with few exceptions. [3]

Value Indistinguishability. Formally, we define the capabilities of $\mathcal{A}$ by a relation $\sim_{\mathcal{A}}$ on values, where $v_1 \sim_{\mathcal{A}} v_2$ means that $\mathcal{A}$ cannot distinguish values v_1 and v_2 in the symbolic model. Specifically, we augment the set of values by fake proofs of the form $\mathrm{SimPr}_\phi(R; v_{1:n})$ (introduced shortly) and define $\sim_{\mathcal{A}}$ to be the smallest relation satisfying:

(i) For any v (which may also be an encryption): $v \sim_{\mathcal{A}} v$

(ii) For any $m, m', b \notin \mathcal{A}$, and $R \neq R'$ with $R, R' \in \mathsf{R}_{\mathsf{hon}}$:
$$\mathrm{Enc}(m, R, \mathrm{Pk}(b)) \sim_{\mathcal{A}} \mathrm{Enc}(m', R', \mathrm{Pk}(b))$$

(iii) For any m, $a \in \mathcal{A}$, and $R, R' \in \mathsf{R}_{\mathsf{hon}}$:
$$\mathrm{Enc}(m, R, \mathrm{Pk}(a)) \sim_{\mathcal{A}} \mathrm{Enc}(m, R', \mathrm{Pk}(a))$$

(iv) For any $\phi, v_{1:n}, v'_{1:m}$ such that $\phi(v_{1:n}, v'_{1:m}) = 1$ and $R \neq R'$ with $R, R' \in \mathsf{R}_{\mathsf{hon}}$:
$$\mathrm{Proof}_\phi(R; v_{1:n}; v'_{1:m}) \sim_{\mathcal{A}} \mathrm{SimPr}_\phi(R'; v_{1:n})$$

Rule (i) simply models that $\mathcal{A}$ cannot distinguish identical values. Rule (ii) models a randomized public key encryption scheme where $\mathcal{A}$ can not distinguish encryptions under different honest randomness but identical public key of an honest account. [4] This rule requires the encryption scheme to hide the length of the encrypted plaintext, which can be achieved by an appropriate padding scheme. Further, the rule implicitly assumes that $\mathcal{A}$ can never learn (a) any private keys of honest accounts, and (b) any honest randomness in $\mathsf{R}_{\mathsf{hon}}$. This assumption is justified: because honest accounts respect the transformation of §5, in any trace (a) no such private keys appear, and (b) honest randomness only occurs in the position of encryption or NIZK proof randomness. Rule (iii) states that the adversary cannot distinguish two fresh encryptions of the same message for the adversary by honest accounts.

The rule (iv) models the zero-knowledge property of NIZK proofs. First, we introduce symbolic fake proofs of the form $\mathrm{SimPr}_\phi(R; v_{1:n})$ for a proof circuit ϕ, randomness R and public arguments $v_{1:n}$. Then, in rule (iv), we define valid proofs generated with honest randomness to be indistinguishable from simulated proofs for the same proof circuit and public arguments. [5] Intuitively, the existence of an indistinguishable fake proof $\mathrm{SimPr}_\phi(R; v_{1:n})$, which is independent of the private arguments $v'_{1:m}$, captures the fact that $\mathrm{Proof}_\phi(R; v_{1:n}; v'_{1:m})$ does not leak any information about $v'_{1:m}$.

Trace Indistinguishability. Next, we define indistinguishability for traces. Two public trace entries v_1@`all` and v_2@`all` are indistinguishable for $\mathcal{A}$, denoted v_1@`all` $\sim_{\mathcal{A}}$ v_2@`all`, if $v_1 \sim_{\mathcal{A}} v_2$. Two public traces t_1 and t_2 are indistinguishable if they (i) are entry-wise indistinguishable, and (ii) have consistent repetition

[3] Our notion is stronger than standard Dolev-Yao-style knowledge deduction rules. For example, in our model the attacker *can* distinguish encrypted values from randomness, which is usually not possible in the latter model [5].

[4] This rule can be viewed as a symbolic model of the standard IND-CPA property of (randomized) public key encryption schemes.

[5] This can be viewed as a symbolic model of the standard zero-knowledge property, which is defined by the existence of a simulator generating such fake proofs [26].

patterns. To understand (ii), consider traces $t_1 = a, a$ and $t_2 = b, c$ with $a \sim_{\mathcal{A}} b$ and $a \sim_{\mathcal{A}} c$ for $b \neq c$. Now, $\mathcal{A}$ can distinguish t_1 from t_2 by checking if the two entries of the trace are identical.

Formally, $t_1 \sim_{\mathcal{A}} t_2$ iff (i) t_1 and t_2 have equal length, and (ii) there exists a bijection π on values such that $\forall i.\ \pi(t_1^i) = t_2^i \wedge \pi(t_1^i) \sim_{\mathcal{A}} t_1^i$, where t^i is the i-th entry of a trace t.

6.2 Observable Information in the Ideal World

We now describe which parts of traces of transactions on a contract C an attacker $\mathcal{A}$ can read if C is executed in an ideal world. From now on, we assume that $\mathcal{A}$ contains **all**, reflecting that the attacker can always access public trace entries.

The *observable trace* describes what parts of a trace are leaked to $\mathcal{A}$ in an ideal world. For a trace t and an attacker $\mathcal{A}$, the observable trace of t, $\mathrm{obs}_{\mathcal{A}}(t)$, is obtained by (i) hiding all values in t whose privacy level is not in $\mathcal{A}$ using a placeholder $\square$, and (ii) dropping all privacy levels. For example, for $t = 1@0x0, 2@0x1, 3$ and $\mathcal{A} = \{\mathbf{all}, 0x0\}$, it is $\mathrm{obs}_{\mathcal{A}}(t) = 1, \square, 3$.

6.3 Data Privacy

Let $\overline{C}$ be the transformation of a well-typed contract C. Intuitively, $\overline{C}$ is private w.r.t. C iff transactions on $\overline{C}$ in the real world do not leak more information than transactions on C in the ideal world. Consider an arbitrary state $\overline{\sigma}$ originating from a sequence of transactions on $\overline{C}$ and let σ be the equivalent state in C (σ exists by Thm. 1 and is unique). $\overline{C}$ is private w.r.t. C iff for any attacker $\mathcal{A}$, there exists a simulator Sim who can, for any transaction T' on $\overline{C}$ under $\overline{\sigma}$ yielding trace $\overline{t}$, produce a trace $\overline{t}_2$ indistinguishable from $\overline{t}$. Sim only has access to $\overline{\sigma}$ and information observed by $\mathcal{A}$ in the ideal world. It has the same capabilities as $\mathcal{A}$ (*e.g.,* it can encrypt values under any public key, but cannot break honest encryption for honest accounts), with two exceptions: it can generate fake proofs (see the definition of $\sim_{\mathcal{A}}$) and *fresh* honest randomness $r \in \mathsf{R}_{\mathsf{hon}}$.[6]

The intuitive idea is that if such a simulator exists, then $\mathcal{A}$ could have simulated $\overline{t}$ herself, without any knowledge of the private data protected by C. Hence, this data is protected by $\overline{C}$. We treat the case of honest and dishonest callers separately, because in the latter case, the transaction may not be the result of a transformation.

Honest Caller. If transaction T' is issued by an honest caller $a \notin \mathcal{A}$, then T' is the transformation of a transaction T in C not throwing an exception (honest callers adhere to §5.4). Assuming the attacker already knows the current state $\overline{\sigma}$, Sim constructs $\overline{t}_2$ from $\overline{\sigma}$, the observable trace of T and the contract code C. This expresses that the attacker learns nothing new from $\overline{t}$ than what he can learn from the code in C and the observable trace of T.

Definition 1 (Privacy for Honest Callers). Contract $\overline{C}$ is private w.r.t. C for honest callers iff for all attackers $\mathcal{A}$, there exists a simulator Sim such that for all $\sigma, \overline{\sigma}$ as defined above the following holds: if $\langle T, \sigma\rangle \overset{t}{\Rightarrow} \langle \sigma', v\rangle$ for some $T, t, \sigma', v \neq \mathbf{fail}$, and $\langle \overline{T}, \overline{\sigma}\rangle \overset{\overline{t}}{\Rightarrow} \langle \overline{\sigma}', \overline{v}\rangle$ for some $\overline{t}, \overline{\sigma}', \overline{v}$, with $\overline{T}$ being the transformation of T, then $\overline{t} \sim_{\mathcal{A}} \overline{t}_2$ for $\overline{t}_2 = \mathrm{Sim}(\overline{\sigma}, \mathrm{obs}_{\mathcal{A}}(t), C)$.

[6] If Sim could only generate dishonest randomness, the attacker could always distinguish encryptions produced by the simulator from honest encryptions. Note that Sim's randomness is fresh, meaning that it is not used by any honest account.

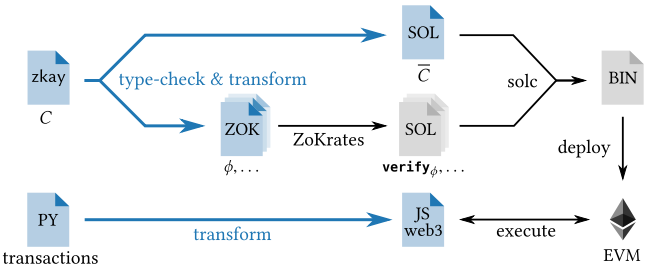


Figure 7: Overview of implementation.

Dishonest Caller. If the caller a is dishonest (*i.e.,* $a \in \mathcal{A}$), T' might not be a transformed transaction. Therefore, the simulator constructs $\overline{t}_2$ only from $\overline{\sigma}$, T' and the code in C. This expresses that the attacker cannot learn any information by crafting arbitrary transactions.

Definition 2 (Privacy for Dishonest Callers). $\overline{C}$ is private w.r.t. C for dishonest callers iff for any attacker $\mathcal{A}$, there exists a simulator Sim′ such that, if running a transaction T' generated by the attacker (*i.e.,* using only private keys from accounts in $\mathcal{A}$) yields $\langle T', \overline{\sigma}\rangle \overset{\overline{t}}{\Rightarrow} \langle \overline{\sigma}', \overline{v}\rangle$, then $\overline{t} \sim_{\mathcal{A}} \overline{t}_2$ for $\overline{t}_2 = \mathrm{Sim}'(\overline{\sigma}, T', C)$.

Privacy Theorem. Thm. 2 states that our transformation respects privacy according to Def. 1 and Def. 2.

Theorem 2. The contract $\overline{C}$ transformed from a well-typed contract C according to §5 is private w.r.t. C for honest and dishonest callers.

We provide a proof of Thm. 2 in App. D. It is based on the fact that the simulator can follow the control flow (which is public) in both C and $\overline{C}$, and simulate encrypted values and NIZK proofs by indistinguishable ciphertexts and fake proofs, respectively.

7 IMPLEMENTATION

We have instantiated our approach for transforming zkay to Solidity contracts in a proof-of-concept implementation using roughly 3 500 lines of Python code.[7] As shown in Fig. 7, our tool type-checks and transforms zkay contracts to Solidity contracts executable on Ethereum, compiling the proof circuits using ZoKrates (discussed shortly). Further, it transforms zkay transactions specified in Python and produces JavaScript code executing the transformed transactions using the web3.js API [25]. In the following, we discuss how to instantiate our approach for Solidity.

NIZK Proofs. We use ZoKrates [23] (commit `224a7e6` with proving scheme `GM17` [30]) to generate Solidity code verifying NIZK proofs. ZoKrates allows representing a proof circuit ϕ in its custom circuit language and generates a verification contract for ϕ. We instantiate $\mathbf{verify}_\phi$ statements as calls to such contracts, and use ZoKrates to generate proofs during transformation of transactions. Some operations (*e.g.,* integer division) are currently not supported by ZoKrates. We reflect these restrictions in zkay's type system by disallowing such operations for private values. We still support them for public values by modifying T_ϕ (Fig. 5f) to use *in* for such

[7] The code is available on GitHub: https://github.com/eth-sri/zkay

Contract	Domain	Short description	#fns (priv)	#loc	#loc transf (sol+zok)	#crossings	scenario #tx (priv)
Exam	Teaching	Record and grade exam answers	4 (3)	37	194 (99 + 95)	15	7 (6)
Income	Welfare	Decide eligibility for welfare programs	4 (3)	23	162 (75 + 87)	8	5 (4)
Insurance	Insurance	Insure secret items at secret amounts	8 (6)	79	341 (159 + 182)	24	9 (7)
Lottery	Gambling	Place bets and claim winnings	4 (4)	37	200 (88 + 112)	7	5 (5)
MedStats	Healthcare	Record medical statistics on patients (Fig. 1)	3 (3)	22	174 (82 + 92)	13	5 (5)
PowerGrid	Energy	Track consumed energy	4 (3)	23	159 (71 + 88)	7	4 (3)
Receipts	Retail	Track and audit cash receipts	4 (4)	28	206 (89 + 117)	11	7 (7)
Reviews	Academia	Blind paper reviews and acceptance decisions	4 (3)	39	198 (104 + 94)	17	6 (5)
SumRing	Security	Simple multi-party computation	3 (3)	23	160 (74 + 86)	8	5 (5)
Token	Finance	Buy and transfer secret amount of tokens	5 (4)	37	234 (112 + 122)	17	6 (5)

Table 1: Evaluated contracts and scenarios. The table specifies for each contract: number of (private) functions, lines of code (loc), lines of code of the transformation result (split into actual contract and ZoKrates proof circuit), number of calls to *in* and *out* during transformation (crossings), and number of (private) transactions in the evaluated scenario.

expressions and computing them outside the proof circuit. For example, the public division `x@all/2` is computed in the contract and its result passed as an argument to the proof circuit ϕ.

Mapping zkay ***Types to ZoKrates Types***. ZoKrates only supports computations on integers in $[0, p-1]$ for a large prime p. By directly mapping the zkay **`uint`** type to ZoKrates integers, we retain correctness in absence of over- and underflows. We encode boolean values **`false`** and **`true`** consistently as numbers 0 and 1, respectively. Encryptions (type **`bin`**, see next) are encoded as integers. Our implementation currently does not support other primitive types within proof circuits.

Encryption. While ZoKrates is considering adding support for asymmetric encryption, its current version does not yet support it. [8] Therefore, we currently use the (insecure) surrogate functions for encryption ($\text{Enc}(v, R, k) = v + k$) and decryption ($\text{Dec}(c, k) = c - k$).

We note that our results will not significantly change once ZoKrates supports asymmetric encryption, because the verification gas cost is essentially independent of the encryption function, and the off-chain cost for proof generation can only grow moderately (we provide a quantitative estimate in §8). This is because our implementation applies a standard reduction [27] to the construction used by ZoKrates, allowing linear-time proof generation (in the size of the circuit), and *constant-cost* verification (after much cheaper hashing of public circuit arguments).

Because Ethereum does not provide a built-in public key infrastructure (PKI), we implemented a simple PKI contract C_{pki} containing a public key storage and providing setter (and getter) functions to announce (resp. retrieve) public keys. [9]

8 EVALUATION

In the following, we demonstrate that our approach is feasible and practical. Specifically, we address the following research questions:

- **Q1** Can zkay express interesting real-world contacts?
- **Q2** What is the development complexity reduction when using zkay compared to using NIZK proofs directly?
- **Q3** What are the (gas) costs for executing transformed contracts on Ethereum?
- **Q4** What are the off-chain costs for transforming contracts and transactions?

Q1: Expressivity of zkay. To showcase the expressivity of zkay, we implemented 10 example contracts, described in Tab. 1 (our implementation contains the full contracts[10]). Our contracts span a wide range of domains such as healthcare, energy, and gambling. While there is active interest in developing blockchain contracts for these domains [33, 41], privacy concerns are a key roadblock preventing their adoption [47].

When implementing the contracts in Tab. 1, zkay helped us to cleanly capture our privacy intents. In our experience, zkay's privacy annotations are a natural way of expressing privacy constraints. Further, zkay's type system is essential: for instance, while developing our examples, we occasionally had to add non-obvious declassifications. Our design choice of explicit declassification and implicit classification is reasonable: while forcing developers to think about leaks, it does not restrict development in the absence of leaks.

Overall, we conclude that zkay is expressive enough to capture a rich class of applications and that programming in zkay is natural.

Q2: Complexity Reduction. We now demonstrate that zkay significantly reduces development complexity compared to using cryptographic primitives directly. To this end, we transform all example contracts and compare the number of lines of the originals with the transformed versions (consisting of Solidity and ZoKrates code). Tab. 1 shows that the number of lines increases significantly, on average by a factor of more than 6. We provide the full output for `MedStats` (Fig. 1a) with our implementation. [11]

Note that because our implementation is not optimized and introduces boiler-plate code, the number of generated code lines can be misleading. Hence, we also evaluate a more specific complexity metric. This is, we investigate the number of times our transformation crosses the boundary between contract code and proof circuit. These crossings happen whenever transformation calls *in* or *out*, or processes a private argument. The number of crossings is critical for development complexity: crossing the boundary is highly error-prone due to the logic being scattered over the contract, proof circuit, and off-chain computation. In particular, crossing the boundary generally requires non-local modifications such as adding statements and arguments to both the function and the proof circuit (as performed by *in* and *out* in Fig. 5). We note that it is possible

[8] https://github.com/Zokrates/ZoKrates/issues/276
[9] https://github.com/eth-sri/zkay/tree/ccs2019/src/compiler/privacy/pki.sol
[10] https://github.com/eth-sri/zkay/tree/ccs2019/eval-ccs2019/examples
[11] https://github.com/eth-sri/zkay/tree/ccs2019/eval-ccs2019/examples/med-stats/reference-compilation

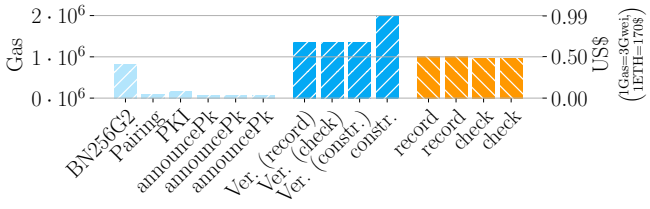


Figure 8: On-chain cost for contract `MedStats` (Fig. 1).

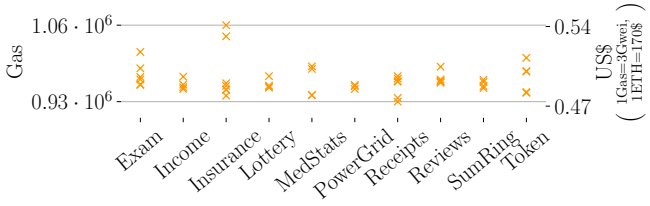


Figure 9: On-chain cost for transformed private transactions.

to reduce code complexity for some crossings. For example, if private variable x is read twice and not modified between the reads, we could pass it to the circuit only once. However, as such optimizations require static insights, they cannot substantially reduce *development* complexity.

Our results in Tab. 1 (column #crossings) indicate that even for simple contracts such as MedStats (see Fig. 1a), many crossings are performed by our tool. Hence, while programming in zkay is deceptively easy, finding errors in transformed contracts is quite hard. Overall, we observe almost 0.4 boundary crossings per line of zkay code, even though many lines do not have functionality (due to empty lines, etc.). We do not expect that the number of crossings can be significantly reduced for our examples.

Q3: On-chain Cost of Privacy. In the following, we discuss the on-chain cost for executing transformed contracts. This cost is measured in terms of gas and is particularly relevant as it directly corresponds to monetary costs paid by the sender of a transaction.

To evaluate this cost, we compile the transformed example contracts, deploy them to a simulated Ethereum blockchain using the truffle suite [19], and execute small transaction scenarios (last column in Tab. 1). [12] All experiments were run on a machine with 32GB RAM and 12 cores at 3.70GHz. We note that on-chain costs only depend on the contract code, not on the simulating machine.

We first discuss the on-chain cost for the `MedStats` contract (Fig. 1), depicted in Fig. 8. Later, we will show how our observations generalize to the other examples. Fig. 8 distinguishes three phases. The first phase ( ) prepares infrastructure required by all contracts. This includes deploying libraries required by ZoKrates (BN256G2, Pairing), and our PKI (PKI, announcePk). Since this is a global one-time task, its (moderate) cost is mostly irrelevant.

The second phase ( ) deploys the contract itself (constr.) and a verifier contract for every private function. This phase only occurs once per deployed contract, and its cost is in the same order of magnitude as the cost of transactions (a repeated cost, discussed shortly). Hence, compared to the cost of transactions, the cost of contract deployment is negligible.

For the final phase ( ), we have implemented a specific scenario where the hospital calls `record` for two different patients who later call `check`. These transaction costs are most relevant, as they occur many times. Fig. 8 shows that every transaction costs roughly 10^6 gas. We believe this is a moderate cost for protecting critical data: at the time of this writing, this corresponded to roughly 0.50 US$. This cost is close to optimal, as it is dominated by the cost of proof verification: Even the (trivial) verification that a private value x equals 0 costs about $0.84 \cdot 10^6$ gas. We note that proof verification costs may be reduced in the future (*e.g.*, using more efficient proof constructions, as discussed in [23]).

The above observations are general: Fig. 9 shows private transaction costs of example scenarios for all contracts. Across all examples, the cost is roughly 10^6 gas. Overall, we conclude that running transactions on transformed contracts is feasible at a moderate cost.

Q4: Off-chain Cost of Privacy. For all our examples, contract transformation took less than 5 minutes, where more than 99% of the total time is due to verifier generation in ZoKrates. Likewise, transforming transactions for our example scenarios took less than 1 minute per transaction, and again, more than 99% of this time is due to proof generation in ZoKrates.

We expect the off-chain computation overhead of real encryption over our surrogate encryption during proof generation to be moderate for each transaction in our evaluation. To estimate this overhead, we used Jsnark with proving scheme `GM17` (as for ZoKrates) to prove that RSA encryption of a message using a 2048-bit key yields a given ciphertext. [13] Generating this proof took roughly 13 seconds, and our generated circuits never used more than 9 encryptions or decryptions.

9 DISCUSSION

In this section, we discuss possible extensions of zkay.

Ownership Transfer. zkay prevents ownership transfer by requiring owner fields to be declared **`final`**. Allowing writing to an owner field `id` outside the constructor would require statically determining all locations owned by `id` and forcing the caller to provide new encryptions (under the new owner's public key) of these locations at modification time, along with an appropriate NIZK proof. Unfortunately, this is not possible if `id` owns entries of dynamically growing mappings (*e.g.*, in **`mapping(uint => uint`**`@id`**`)`**) since the set of locations owned by `id` cannot be determined at compile time in this case. Still, zkay could be easily extended to support mutability of fields not owning array entries.

Function Calls. Extending zkay to support calls to fully public functions is straight-forward. Handling calls to non-recursive private functions with statically known body is also possible, by tunneling arguments induced by the transformation of the callee (such as proofs) through the caller. We note that recursive functions are rare in Solidity contracts, as they quickly exceed the gas limit.

Alternative Realizations. This work leverages encryptions and NIZK proofs to realize zkay contracts. However, we stress that

[12] We use version 0.5.0 of solc, 5.0.14 of Truffle, and 2.5.5 of Ganache.

[13] Using PKCS#1 v2.2, following https://github.com/akosba/jsnark/blob/master/JsnarkCircuitBuilder/src/examples/gadgets/rsa/RSAEncryptionOAEPGadget.java

zkay's interpretable syntax together with its intuitive enforced privacy notion is largely independent of its realization. Thus, zkay is not fundamentally restricted to NIZK proofs on encrypted states, but could also leverage other building blocks, such as partially or fully homomorphic encryption, trusted hardware, or NIZK proofs on hashed states. Thus, zkay can be seen as part of a broader effort to lift realizations using low-level building blocks to high-level specifications.

Setup Phase. Current NIZK proof constructions compatible with Ethereum [29, 30, 42] rely on a trusted setup phase (once per circuit)—a known deployment issue [14]. However, secure multi-party computation (SMC) can be used to reduce trust for this setup phase [8], which is only required once per contract. Alternatively, there exist recent proof constructions (*e.g.,* zk-STARKs [6]) without a trusted setup phase.

Compatibility with Existing Analysis Tools. Various tools enable verification, testing, and other analysis of smart contracts [32, 48, 52]. Applying them to zkay is possible, as dropping privacy annotations from a zkay specification contract (*i.e.,* before transformation) results in a fully public contract whose functionality can be checked by existing tools. However, note that properties affected by the transformation (*e.g.,* gas cost) can only be verified in the transformed contracts.

10 RELATED WORK

We now discuss the works that are most closely related to ours.

Blockchain Privacy. Many works bring privacy to payments and transactions using NIZK proofs [38, 44], custom primitives [17], or off-chain payment channels [28, 31]. In contrast to all these approaches, we address data privacy for general smart contracts.

Like us, Hawk [35], Arbitrum [34], and Ekiden [18] provide data privacy for general smart contracts without complicating contract development. However, they all rely on trusted managers or hardware. Concretely, Hawk and Arbitrum trust managers for privacy (but not for correctness), meaning a compromised manager can disclose users' private data—a substantial risk, as data leaks are hard to detect and even harder to prevent. We note that replacing trusted managers by secure multi-party computation (SMC) or trusted execution environments (TEEs) introduces scalability issues for SMC (discussed shortly) and new attack vectors for TEEs (cp. Ekiden). Ekiden leverages trusted hardware in the form of TEEs. However, if an attacker can forge attestation reports for a small set of K TEEs (a practical attack [49]), she can violate the correctness of contracts computations. In contrast to these approaches, zkay only relies on cryptographic primitives and thus provides stronger security guarantees.

Zexe [14] introduces a decentralized private computation scheme. Unfortunately, it uses non-standard function specification primitives (so-called *predicates*) and does not discuss how these can for example incorporate unbounded data structures or loops. In contrast, our approach provides an explicit function encoding supporting dynamic mappings and public loops.

Secure Multi-Party Computation. SMC hides the input of all parties involved in a computation, providing data privacy by construction. In practice however, SMC cannot scale to the number of participants in public blockchains—recent SMC systems handle only up to 150 parties [4, 51].

Still, zkay shares some aspects with high-level languages for specifying SMC, such as restrictions on control flow and loops. However, a key contribution of zkay is its compilation to NIZK proofs (in contrast to SMC-based compilation).

Moreover, as we demonstrate in the following, while zkay's type system and privacy notion seem superficially similar to SMC-based languages, they are technically different due to various blockchain-specific challenges. First, SMCL [40], Wysteria [43], SecreC [13], and ObliVM [36] distinguish public values (in cleartext) from private values (readable by collaboration of multiple participants). In contrast, while zkay also distinguishes public from private values, the latter are private to their *owner*, the only participant who can read them (a restriction reflected in zkay's type system, see C2). We note that SMCL and Wysteria support values stored at only one participant, but subject to tampering. Second, SecreC and ABY [21] annotate variables with privacy types indicating the protocol used to represent and operate on this variable. In contrast, while zkay also supports privacy types, they specify the variable's owner. Third, Fairplay [37], SecreC, and ObliVM do not support general loops with private conditions. In contrast, zkay must additionally disallow private computations in loop bodies. Fourth, like zkay, SMCL and SecreC specify the expected leakage of computations by ideal-world traces. However, zkay's ideal-world traces are more fine-grained, as they depend on the owner of variables.

Zero-Knowledge Statements as Programs. In TinyRAM [7, 9, 10], NP statements can be expressed as programs (instead of circuits) and efficiently verified in zero-knowledge. Compared to zkay, TinyRAM also provides a form of data privacy (by hiding the NP witness), but lacks a privacy type system and a blockchain-specific privacy notion. While conceptually, zkay's proof circuits could be expressed as TinyRAM programs, there is currently no tool support for their verification on Ethereum, confining us to using ZoKrates.

Privacy Policy Languages. JFlow [39] introduces information flow annotations enforcing fine-grained and powerful access control. Jeeves [2, 55] and Jacqueline [54] are languages separating core logic from non-interference policy specifications.

All three languages are substantially different from zkay as their execution model assumes a trusted system enforcing these policies.

11 CONCLUSION

We presented zkay, a typed language using privacy types to specify owners of private values. To enable running a zkay contract on public blockchains, we transform it to a contract where values are encrypted for their owner and correctness is enforced using NIZK proofs, guaranteeing that transformed contracts preserve privacy and functionality w.r.t. the specification contract. Solving four key challenges when using NIZK proofs, our language disallows contracts that cannot be realized (C1, C2), allows intuitively specifying the contract's logic (C3), and prevents implicit leaks (C4).

Our evaluation shows that transformed contracts are runnable on the Ethereum blockchain at a moderate cost. Our approach demonstrates that automatic compilation of high-level privacy specifications to low-level primitives for smart contracts is possible, setting the stage for more research in this area.

REFERENCES

[1] N. Z. Aitzhan and D. Svetinovic. 2018. Security and Privacy in Decentralized Energy Trading Through Multi-Signatures, Blockchain and Anonymous Messaging Streams. *IEEE Transactions on Dependable and Secure Computing* (2018).

[2] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-agnostic Programs. In *PLAS'13*. https://doi.org/10.1145/2465106.2465121

[3] Michael Backes and Dominique Unruh. 2008. Computational Soundness of Symbolic Zero-Knowledge Proofs Against Active Attackers. In *CSF '08*. https://doi.org/10.1109/CSF.2008.20

[4] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. 2018. An End-to-End System for Large Scale P2P MPC-as-a-Service and Low-Bandwidth MPC for Weak Participants. In *CCS '18*. https://doi.org/10.1145/3243734.3243801

[5] David Basin, Jannik Dreier, and Ralf Sasse. 2015. Automated Symbolic Proofs of Observational Equivalence. In *CCS '15*. https://doi.org/10.1145/2810103.2813662

[6] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive. (2018). https://eprint.iacr.org/2018/046.

[7] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *CRYPTO '13*.

[8] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. 2015. Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs. In *SP '15*. https://doi.org/10.1109/SP.2015.25

[9] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-interactive Zero Knowledge for a Von Neumann Architecture. In *SEC'14*. http://dl.acm.org/citation.cfm?id=2671225.2671275

[10] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2017. Scalable Zero Knowledge Via Cycles of Elliptic Curves. *Algorithmica* 4 (Dec. 2017). https://doi.org/10.1007/s00453-016-0221-0

[11] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2012. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS '12*. https://doi.org/10.1145/2090236.2090263

[12] Manuel Blum, Paul Feldman, and Silvio Micali. 1988. Non-interactive zero-knowledge and its applications. In *STOC '88*. https://doi.org/10.1145/62212.62222

[13] Dan Bogdanov, Peeter Laud, and Jaak Randmets. 2014. Domain-Polymorphic Programming of Privacy-Preserving Applications. In *PLAS'14*. https://doi.org/10.1145/2637113.2637119

[14] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2018. Zexe: Enabling Decentralized Private Computation. *IACR Cryptology ePrint Archive* (2018).

[15] Vitalik Buterin. 2016. Privacy on the Blockchain. Available from: https://blog.ethereum.org/2016/01/15/privacy-on-the-blockchain/.

[16] Vitalik Buterin and Christian Reitwiessner. 2017. EIP 197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128. http://eips.ethereum.org/EIPS/eip-197 Accessed 2019-04-11.

[17] Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed Kosba, Ari Juels, and Elaine Shi. 2017. Solidus: Confidential Distributed Ledger Transactions via PVORM. In *CCS '17*.

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

[19] ConsenSys. 2018. Truffle Suite. https://truffleframework.com/.

[20] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL '77*. https://doi.org/10.1145/512950.512973

[21] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS '15*. https://doi.org/10.14722/ndss.2015.23113

[22] D. Dolev and A. C. Yao. 1981. On the security of public key protocols. In *SFCS '81*. https://doi.org/10.1109/SFCS.1981.32

[23] Jacob Eberhardt and Stefan Tai. 2018. ZoKrates - Scalable Privacy-Preserving Off-Chain Computations. In *IEEE International Conference on Blockchain*. http://www.ise.tu-berlin.de/fileadmin/fg308/publications/2018/2018_eberhardt_ZoKrates.pdf

[24] Ethereum Foundation. 2018. Solidity Documentation. https://solidity.readthedocs.io/en/v0.4.24/. Accessed: 2018-08-23.

[25] Ethereum Foundation. 2019. web3.js Ethereum JavaScript API. https://github.com/ethereum/web3.js. Accessed 2019-05-07.

[26] Uriel Feige, Dror Lapidot, and Adi Shamir. 1999. Multiple NonInteractive Zero Knowledge Proofs Under General Assumptions. *SIAM J. Comput.* 1 (Jan. 1999). https://doi.org/10.1137/S0097539792230010

[27] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. 2013. Quadratic Span Programs and Succinct NIZKs without PCPs. In *EUROCRYPT '13*. https://doi.org/10.1007/978-3-642-38348-9_37

[28] Matthew Green and Ian Miers. 2017. Bolt: Anonymous Payment Channels for Decentralized Currencies. In *CCS '17*. https://doi.org/10.1145/3133956.3134093

[29] Jens Groth. 2016. On the Size of Pairing-based Non-interactive Arguments. Cryptology ePrint Archive, Report 2016/260. https://eprint.iacr.org/2016/260.

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

[31] Ethan Heilman, Foteini Baldimtsi, and Sharon Goldberg. 2016. Blindly Signed Contracts: Anonymous On-Blockchain and Off-Blockchain Bitcoin Transactions. (2016).

[32] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. 2018. KEVM: A Complete Semantics of the Ethereum Virtual Machine. In *CSF'18*. IEEE.

[33] Harry Hodges. 2017. Medical data and the rise of blockchain. Available from: https://www.bookingbug.com/blog/5-ways-blockchain-will-change-the-face-of-retail/. Accessed: 2019-05-11.

[34] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. 2018. Arbitrum: Scalable, private smart contracts. In *USENIX Security*.

[35] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. 2016. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *SP '16*. https://doi.org/10.1109/SP.2016.55

[36] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *SP '15*. https://doi.org/10.1109/SP.2015.29

[37] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay—a Secure Two-party Computation System. In *SSYM'04*. http://dl.acm.org/citation.cfm?id=1251375.1251395

[38] I. Miers, C. Garman, M. Green, and A. D. Rubin. 2013. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *SP '13*. https://doi.org/10.1109/SP.2013.34

[39] Andrew C. Myers and Andrew C. Myers. 1999. JFlow: Practical Mostly-static Information Flow Control. In *POPL '99*. https://doi.org/10.1145/292540.292561

[40] Janus Dam Nielsen and Michael I. Schwartzbach. 2007. A domain-specific programming language for secure multiparty computation. In *PLAS'07*. https://doi.org/10.1145/1255329.1255333

[41] Mike Orcutt. 2017. How Blockchain Could Give Us a Smarter Energy Grid. Accessed: 2019-05-09. Available from: https://www.technologyreview.com/s/609077/how-blockchain-could-give-us-a-smarter-energy-grid/.

[42] B. Parno, J. Howell, C. Gentry, and M. Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *SP '13*. https://doi.org/10.1109/SP.2013.47

[43] A. Rastogi, M. A. Hammer, and M. Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *SP '14*. https://doi.org/10.1109/SP.2014.48

[44] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *SP '14*. https://doi.org/10.1109/SP.2014.36

[45] Benedikt Scheungraber. 2018. Eliminate the hassle of flight delay compensation by using smart contracts. Available from: https://medium.com/cashlink-crypto/eliminate-the-hassle-of-flight-delay-compensation-by-using-smart-contracts-a5db3b5c3ed.

[46] Muyao Shen. 2018. Crypto Valley Declares Blockchain Voting Trial a 'Success'. Available from: https://www.coindesk.com/crypto-valley-declares-blockchain-voting-trial-a-success/.

[47] Lydia Torne and Sophie Sheldon. 2018. Medical data and the rise of blockchain. Available from: http://www.pharmatimes.com/web_exclusives/medical_data_and_the_rise_of_blockchain_1243441. Accessed: 2019-05-09.

[48] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *CCS'18*. https://doi.org/10.1145/3243734.3243780

[49] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *USENIX Security*.

[50] Kevin Wang and Ali Safavi. 2017. Blockchain is empowering the future of insurance. Available from: https://techcrunch.com/2016/10/29/blockchain-is-empowering-the-future-of-insurance/.

[51] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Global-Scale Secure Multiparty Computation. In *CCS '17*. https://doi.org/10.1145/3133956.3133979

[52] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. 2018. Formal Specification and Verification of Smart Contracts for Azure Blockchain. *arXiv:1812.08829 [cs]* (Dec. 2018). http://arxiv.org/abs/1812.08829

[53] Gavin Wood. 2016. Ethereum: a Secure Decentralised Generalised Transaction Ledger.

[54] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-backed Applications. In *PLDI '16*.

[55] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A Language for Automatically Enforcing Privacy Policies. In *POPL '12*. https://doi.org/10.1145/2103656.2103669

(External) Account	Account owned by a human user
Address	Value identifying an account
Caller	Account calling a contract function
Transaction	Contract function call by external account
Off-chain computation	Locally on machine of caller
On-chain computation	Publicly validated on blockchain
Secret arguments	Values only known by the prover (see below)
Public arguments	Public values parameterizing proof circuits
Proof statement/circuit	Constraint on secret and public arguments
Proof	Evidence that some secret args satisfy circuit
Zero-knowledge	Only existence of secret arguments is leaked
Prover (Verifier)	Algorithm generating (verifying) proofs

Figure 10: Brief summary of blockchain and NIZK terminology.

$$\frac{c \in [\![\tau]\!]}{\Gamma \vdash c\colon \tau@\texttt{all}} \qquad \frac{\Gamma \vdash e\colon \tau@\texttt{me}}{\Gamma \vdash \texttt{reveal}(e, \alpha)\colon \tau@\alpha} \qquad \frac{\Gamma \Vdash L\colon \tau@\texttt{all}}{\Gamma \vdash L\colon \tau@\texttt{all}}$$

$$\frac{\Gamma \Vdash L\colon \tau@\alpha \quad \alpha \text{ provably evaluates to caller}}{\Gamma \vdash L\colon \tau@\texttt{me}}\ \texttt{priv-read}$$

$$\frac{\vdash g\colon \prod_{i=1}^{n} \tau_i@\alpha_i \to \tau@\alpha \qquad \begin{array}{c}\Gamma \vdash e_1\colon \tau_1@\alpha_1 \\ \cdots \\ \Gamma \vdash e_n\colon \tau_n@\alpha_n\end{array}}{\Gamma \vdash g(e_1, \ldots, e_n)\colon \tau@\alpha}\ \texttt{eval-native}$$

Figure 11: Typing rules for expressions in zkay.

A BLOCKCHAIN AND NIZK TERMINOLOGY

Fig. 10 summarizes important terminology used in this work.

B TYPING RULES IN ZKAY

In this section, we discuss zkay's type system. Because the typing rules to derive data types are standard, we focus on privacy types.

B.1 Format of Typing Rules

We write $\Gamma \vdash e\colon \tau@\alpha$ (resp. $\Gamma \Vdash L\colon \tau@\alpha$) to indicate that expression e (resp. location L) is of type $\tau@\alpha$ under the typing context Γ. We write $\vdash g\colon \prod_{i=1}^{n} \tau_i@\alpha_i \to \tau@\alpha$ to express that the i^{th} argument of a native function g is of type $\tau_i@\alpha_i$, and the return value is of type $\tau@\alpha$, where $\alpha, \alpha_1, \ldots, \alpha_n \in \{\texttt{me}, \texttt{all}\}$. Allowing other privacy types is not desirable, since functions should only be able to read public arguments or arguments private to the caller, and the caller should only be able to read public or self-owned return values. We write $\Gamma \overset{P}{\rightsquigarrow} \Gamma'$ to indicate that statement P is well-typed and transforms the typing context Γ to Γ', capturing that P might declare new variables and thereby modify the context.

B.2 Typing Rules

Expressions. In general, expressions can only be read if they are public or private to the caller. We still allow expressions with privacy type $\alpha \notin \{\texttt{me}, \texttt{all}\}$, but our type system implicitly ensures that such expressions can only be used as right-hand sides of assignments (*e.g.,* $\texttt{reveal}(10, x)$ for x of type $\texttt{address@all}$).

Expression `me` has type $\texttt{address@all}$. The remaining typing rules for expressions are shown in Fig. 11. The rule for constants c indicates they are always public. Here, $[\![\tau]\!]$ denotes the set of values with type τ. For example, $[\![\texttt{uint}]\!]$ denotes non-negative integers. We provide the formal definition of $[\![\cdot]\!]$ in §C.1.

The next rule describes how $\texttt{reveal}(e, \alpha)$ can be used to reveal an expression e (private to the caller) to an arbitrary privacy type α. This allows explicitly declassifying information to make it public (by setting $\alpha = \texttt{all}$), or reclassifying information for some other owner $\alpha \notin \{\texttt{me}, \texttt{all}\}$. Note that in the latter case, the resulting expression can only be used as a right-hand side of an assignment.

Though the privacy type of locations can be an arbitrary address, when *reading* from a location L, it is crucial that L is readable for the caller. In this case, we treat L as an expression and restrict its privacy type to $\alpha \in \{\texttt{me}, \texttt{all}\}$. If the location is public, the expression based on this location can be annotated as `all`. Otherwise, rule `priv-read` enforces that the location is provably private to the caller. We leverage Abstract Interpretation [20] to check whether α can be proven to evaluate to the same value as `me` at runtime. If so, the rule annotates the expression reading the location as `me`.

The rule `eval-native` for evaluating native functions is standard (we discuss how to type native functions themselves shortly).

Privacy Types. A privacy type α is either an identifier id of type $\texttt{address@all}$, `me`, or `all`. Although `all` is technically not an expression, we also assign it the type $\texttt{address@all}$ for simplicity, meaning that all privacy types are of type $\texttt{address@all}$.

Locations. Fig. 12 shows the typing rules for locations. The type of identifiers is determined by the typing context.

For mappings, in order to avoid passing whole mappings to the proof circuit later, we require mappings themselves and keys into mappings to be public, and only allow individual mapping entries to be private. For a general key type τ, each entry in the mapping must be annotated with the same privacy type α, and reading the entry at key e yields $L[e]$ of privacy type α. For key type `address`, we allow α to contain 'id', enabling the privacy types of the entries of L to depend on the key. When reading L at key e, we syntactically substitute 'id' by e in $\tau@\alpha$. Because 'id' stands for a privacy type, we require e to be either an identifier or `me`.

Statements. The rules for sequential composition and `skip` statements are standard. A statement '$\tau@\alpha$ id' declares a variable of type $\tau@\alpha$, and its typing rule ensures that α in fact evaluates to a public address:

$$\frac{\Gamma \vdash \alpha\colon \texttt{address@all} \quad \text{id} \notin \Gamma}{\Gamma \overset{\tau@\alpha\ \text{id}}{\rightsquigarrow} \Gamma, \text{id}\colon \tau@\alpha}\ \texttt{decl}$$

We show typing rules for the additional statements in Fig. 12. The typing rule for $\texttt{verify}_\phi$ is included to express transformed contracts and we assume that the original contracts never use a $\texttt{verify}_\phi$ statement. The rule requires that the data types of the provided arguments match the types of the first n arguments of ϕ, and that they are public. The proof circuit ϕ is a function taking $n + m$ arguments, the first n of which are publicly provided, and the remaining m arguments are part of the proof e_0. The data types τ_i, τ_i' are restricted to primitive types (*i.e.,* `bool`, `uint`, `address`, and `bin`) to avoid passing whole mappings to verification.

The typing rule for assignments ensures that the *data* type of the location L is consistent with the right-hand side expression e.

$$\frac{x\colon \tau@\alpha \in \Gamma}{\Gamma \Vdash x\colon \tau@\alpha} \qquad \frac{\Gamma \Vdash L\colon \texttt{mapping}(\tau \texttt{ => } \tau'@\alpha)@\texttt{all} \quad \Gamma \vdash e\colon \tau@\texttt{all}}{\Gamma \Vdash L[e]\colon \tau'@\alpha} \qquad \frac{\Gamma \Vdash L\colon \texttt{mapping}(\texttt{address!id => } \tau@\alpha)@\texttt{all} \quad \Gamma \vdash e\colon \texttt{address@all} \qquad \mathrm{id} \notin \Gamma \quad (e = \mathrm{id}' \ \vee \ e = \texttt{me})}{\Gamma \Vdash L[e]\colon \tau@\alpha[\mathrm{id} \mapsto e]}$$

Figure 12: Typing rules for locations.

$$\frac{\Gamma \Vdash L\colon \tau@\alpha \quad \Gamma \vdash e\colon \tau@\alpha' \quad (\alpha = \alpha' \vee \alpha' = \texttt{all})}{\Gamma \xrightarrow{L=e} \Gamma} \qquad \frac{\Gamma \vdash e\colon \texttt{bool@all}}{\Gamma \xrightarrow{\texttt{require}(e)} \Gamma} \qquad \frac{\phi\colon \left(\prod_{i=1}^{n} [\![\tau_i]\!] \times \prod_{j=1}^{m} [\![\tau'_j]\!]\right) \to \{0,1\} \qquad \tau_i,\ \tau'_j \text{ primitive} \quad \Gamma \vdash e_0\colon \texttt{bin@all} \quad \Gamma \vdash e_1\colon \tau_1\texttt{@all} \quad \cdots \quad \Gamma \vdash e_n\colon \tau_n\texttt{@all}}{\Gamma \xrightarrow{\texttt{verify}_\phi(e_0, e_1, \ldots, e_n)} \Gamma}$$

Figure 13: Typing rules for selected statements in zkay.

We allow the privacy type of L to be different from the privacy type of e only if e is public. Hence, we allow implicit classification of a public value for any owner, but forbid implicit de- or reclassification. Assignments can be used in combination with **reveal** expressions to write explicitly reclassified information.

Functions and Contracts. The return value of native functions $g(e_1, \ldots, e_n)$ is conservatively typed private if at least one of the arguments e_i is private to the caller, and public otherwise. We provide multiple signatures for different argument privacy types. The signatures are of the form $\tau_1@\alpha_1 \times \cdots \times \tau_n@\alpha_n \to \tau@\min(\alpha_1, \ldots, \alpha_n)$, where min returns **all** if and only if all its arguments are **all**, and **me** otherwise. For example, $e_1 \texttt{?}\ e_2 : e_3$ takes a condition (**bool**) and two numbers (**uint**), and returns a number (**uint**). Hence, the following pattern captures all possible signatures of this native function:

$$\texttt{bool@}\alpha_1 \times \texttt{uint@}\alpha_2 \times \texttt{uint@}\alpha_3 \to \texttt{uint@}\min(\alpha_1, \alpha_2, \alpha_3).$$

The data types of native functions are as follows. The public key infrastructure **pk** takes an address (**address**) and returns the public key of that address (**bin**). The rules for unary ($\ominus$) and binary ($\oplus$) arithmetic and boolean expressions are standard.

A function defined in a contract is well-typed if its body P is well-typed in a context including all contract fields and arguments. Like for native functions, its arguments and return value must have a privacy type in {**all**, **me**}.

A contract C is well-typed if all its functions are well-typed (under the context induced by the fields of C), and all privacy annotations of its fields are **all** or public addresses declared as **final**.

C FORMAL SEMANTICS

In this section, we provide formal semantics for zkay.

C.1 Semantics of Types

For each data type τ, we define the set $[\![\tau]\!]$ of values a variable of type τ may assume. For example, $[\![\texttt{uint}]\!] = [0, 2^{32} - 1]$, $[\![\texttt{bool}]\!] = \{\texttt{true}, \texttt{false}\}$, and $[\![\texttt{address}]\!] = \{0,1\}^*$. Further, $[\![\texttt{bin}]\!]$ consists of symbolic representations for keys, ciphertexts, proofs and encryption randomness. Symbolic public and secret keys are of the form $\mathrm{Pk}(a)$ and $\mathrm{Sk}(a)$, for an address a. The semantics of $\texttt{mapping}(\tau_1 \texttt{ => } \tau_2)$ is given by a partial function from $[\![\tau_1]\!]$ to $[\![\tau_2]\!]$. For example, a value of type $\texttt{mapping}(\texttt{uint => uint})$ is $\{1 \mapsto 4, 2 \mapsto 4\}$. Reading a mapping for an uninitialized key yields undefined behavior.

C.2 Semantics of Contexts

A typing context Γ contains typed variables and contract fields. Its semantics $[\![\Gamma]\!]$ describes the set of all possible states with respect to the typed variables and contract fields contained in it. For instance, the context $\Gamma = \{x\colon \texttt{mapping}(\texttt{uint => uint@all}), y\colon \texttt{uint}\}$ contains the state $\sigma = \{x \mapsto \{1 \mapsto 2\}, y \mapsto 2\}$ To update a state σ, we write $\sigma[l \leftarrow v]$ for a runtime location l and value v.

C.3 Semantics of Language Constructs

Fig. 14 summarizes notation for the semantics of language constructs. All executions (i) start from a state $\sigma \in [\![\Gamma]\!]$, where Γ is the typing context at the beginning of the execution, and (ii) produce a trace t. Evaluating a location L yields a runtime location l. Evaluating an expression e yields a value v. Evaluating a function F requires a starting state σ (providing values for contract fields) and function arguments $(v_1, \ldots, v_n)$. It results in an updated state σ' and a return value v. Executing a statement returns a state σ'. Finally, executing a transaction on state σ (providing values for contract fields) results in an updated state σ' (also providing values for contract fields) and a return value v.

Locations	$\langle L, \sigma\rangle \overset{t}{\Vdash\!\!\mapsto} l$
Expressions	$\langle e, \sigma\rangle \overset{t}{\mapsto} v$
Functions	$\langle F, \sigma, v_{1:n}\rangle \overset{t}{\mapsto} \langle \sigma', v\rangle$
Statements	$\langle P, \sigma\rangle \overset{t}{\mapsto} \sigma'$
Transactions	$\langle T, \sigma\rangle \overset{t}{\Rightarrow} \langle \sigma', v\rangle$

Figure 14: Notation for semantics.

Exceptions. Executions in zkay may throw exceptions, captured by setting the right-hand side of the semantic rule to **fail**. For example, the semantics of a division-by-zero expression is $\langle 1/0, \sigma\rangle \overset{1,0,\texttt{fail}}{\longmapsto\!\!\!\!\longrightarrow}$ **fail**. Analogously, we set l, v or σ' to **fail** to indicate exceptions for other constructs. Thrown exceptions stop the execution of a given transaction.

Locations. For location evaluation, the trace t ends with the runtime location annotated with privacy level **all**, reflecting the fact that the location of a write cannot be hidden. Identifiers need not be evaluated further, hence their semantics is $\langle \mathrm{id}, \sigma\rangle \overset{\mathrm{id@all}}{\Vdash\!\!\!\!\longmapsto} \mathrm{id}$. Further, Fig. 15 provides semantics for mapping lookups.

Expressions. Our expressions do not support side-effects (allowing them is straightforward, but would clutter our presentation). In general, the trace t when evaluating expression e contains (as its last entry) the value v resulting from evaluating e, with privacy

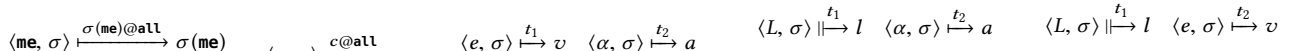


Figure 15: Semantics for selected expressions and locations. For location reads, we assume that L is typed according to Fig. 11.

level a based on the privacy type of e. Assuming the privacy type of e is α, we determine the privacy level a of v by evaluating α.

Fig. 15 provides semantics for selected expressions. While **all** is not technically an expression, providing semantics for it enables us to evaluate privacy types α. For location reads, the rule determines the privacy level a of the value at location l by evaluating α.

Functions. The semantics of native functions $g(e_1, \ldots, e_n)$ is standard (and thus omitted). The trace of the evaluation only consists of the return value and the traces of evaluating the arguments.

Fig. 16 describes the semantics of contract functions. A function specified in contract C (i) keeps only the contract fields and the caller field **me** from the current state σ (indicated by $\sigma[C]$), (ii) extends the resulting state $\sigma[C]$ with all arguments (indicated by $\mathrm{id}_{1:n} \leftarrow v_{1:n}$), (iii) runs the function body P, resulting in state σ', and (iv) evaluates e, resulting in value v. Then, it (v) updates σ with the contract fields from σ' (indicated by $\sigma[C \leftarrow \sigma'[C]]$) and (vi) returns v.

Statements. Fig. 17 shows the semantics of selected statements. For **verify**, it checks whether p is a NIZK proof certifying that there exist private values $v'_{1:m}$ such that ϕ evaluates to 1 when applied to the values of the public expressions $e_1, \ldots, e_n$ and the private values $v'_{1:m}$. An analogous rule (not shown) throws an exception if any of the preconditions does not hold.

The semantics for the remaining statements is mostly straightforward. Sequential composition propagates exceptions to the end of the program. For example, if P_1 fails, we skip P_2, and directly return **fail**. For **require**(e), we leave the state unchanged if e evaluates to **true**. Otherwise, we throw an exception.

Transactions. In a transaction, an account a calls a function f of a contract C, using arguments $v_1, \ldots, v_n$, written as $\mathrm{Tx}^{(a)}_{C.f}(v_1, \ldots, v_n)$.

Fig. 18 shows the semantics of transactions. The rule `T-ok` (i) looks up f in C by $C[f]$, (ii) runs F on the current state (extended with the caller address a stored under **me**) and the provided arguments, resulting in a new state σ' and a return value v and (iii) returns the result v and updates the state to $\sigma'[C]$, only keeping contract fields in C. The resulting trace contains (i) C publicly, (ii) f publicly, (iii) the caller address publicly, (iv) all public arguments, (v) all private arguments, and (vi) the trace of running f (which includes the final return value v). If F throws an exception, it triggers rule `T-fail`, which rolls back the state to the σ and returns **fail**.

D PRIVACY OF TRANSFORMATION

In this section, we prove Thm. 2. We consider privacy for honest and dishonest callers separately. In both cases, the simulator has access to the contract code C and hence to $\overline{C}$ (as the transformation is deterministic). To simplify the proof, we assume that the specification contract C does not perform NIZK proof verification.

D.1 Privacy for Honest Callers

Let $\mathcal{A}$ be an arbitrary attacker. We now construct the simulator Sim for honest caller transactions. Let σ be a state in C that is equivalent to some $\overline{\sigma}$ in $\overline{C}$ as defined in §6 and assume $\langle T, \sigma\rangle \overset{t}{\Rightarrow} \langle \sigma', v\rangle$ for some T, t, σ', $v \neq$ **fail** and $\langle \overline{T}, \overline{\sigma}\rangle \overset{\overline{t}}{\Rightarrow} \langle \overline{\sigma}', \overline{v}\rangle$ for some $\overline{t}$, $\overline{\sigma}'$, $\overline{v}$. We show how Sim constructs a trace $\overline{t}_2$ indistinguishable from $\overline{t}$, given $\overline{\sigma}$, $t^* := \mathrm{obs}_{\mathcal{A}}(t)$ and the contract code of C.

Simulating Encryptions and NIZK Proofs. We first describe how Sim can *simulate* (meaning, produce values indistinguishable from) NIZK proofs and encrypted values occurring in $\overline{t}$ by providing four simulation procedures (S1–S4).

(S1) Encryptions created by honest for honest accounts. Sim simulates encryptions of the form $\mathrm{Enc}(m, R, \mathrm{Pk}(a))$ for some $m, R \in \mathsf{R}_{\mathsf{hon}}$, and $a \notin \mathcal{A}$ by computing $\mathrm{Enc}(0, R', \mathrm{Pk}(a))$ for some fresh honest randomness $R' \in \mathsf{R}_{\mathsf{hon}}$ and constant 0. Because $R' \neq R$ (since R' is fresh), the two values are indistinguishable by (ii) in §6.1.

(S2) Encryptions created by honest for dishonest accounts. Such encryptions have the form $\mathrm{Enc}(m, R, \mathrm{Pk}(a))$ for some m, $R \in \mathsf{R}_{\mathsf{hon}}$ and $a \in \mathcal{A}$. If the simulator knows m, it can simulate the encryption by creating $\mathrm{Enc}(m, R', \mathrm{Pk}(a))$ for some fresh randomness $R' \in \mathsf{R}_{\mathsf{hon}}$. The two values are indistinguishable by (iii) in §6.1.

(S3) Encryptions created by dishonest accounts. Such encryptions have the form $\mathrm{Enc}(m, R, \mathrm{Pk}(a))$ for some m, $R \in \mathsf{R}_{\mathsf{adv}}$ and some address a. Because the caller is honest, such encryptions cannot be part of transaction parameters and only occur in $\overline{t}$ if they are part of $\overline{\sigma}$. Hence, the simulator can simulate these values by copying them from $\overline{\sigma}$. The copied values are indistinguishable by (i) in §6.1.

(S4) NIZK proofs. Given $v_{1:n}$, a valid proof $\mathrm{Proof}_{\phi}(R; v_{1:n}; v'_{1:m})$ can be simulated by constructing $\mathrm{SimPr}_{\phi}(R'; v_{1:n})$ for fresh $R' \in \mathsf{R}_{\mathsf{hon}}$. These proofs are indistinguishable by (iv) in §6.1.

Simulating the Real World Trace. Transforming C to $\overline{C}$ leads to the following structural changes: (i) additional function arguments are introduced, (ii) private composite and declassified expressions are replaced by such function arguments, and (iii) NIZK proof verifications are introduced at the end of each (private) function.

Because all control flow conditions in C are readily available in t^* (they are public), Sim can follow the control flow of transaction T in C. Next, we will show how Sim can simultaneously track the (identical) control flow in $\overline{C}$ and build $\overline{t}_2$ by respecting the three changes (i–iii) described above and simulating encryptions and NIZK proofs using S1–S4.

Creating a trace indistinguishable from $\overline{t}$ requires consistently reproducing repetitions of equal values (see the bijection π in the definition of trace indistinguishability, §6.1). Sim can track the runtime locations accessed in $\overline{C}$ because it knows the code of $\overline{C}$ and all used mapping keys are available in t^* (they are public by the type system). Hence, in the following, Sim can produce consistent

$$\frac{F = \texttt{function}\ \mathrm{id}(\tau_1@\alpha_1\ \mathrm{id}_1, \ldots, \tau_n@\alpha_n\ \mathrm{id}_n)\ \texttt{returns}\ \tau@\alpha\ \{P; \texttt{return}\ e;\} \qquad \langle P, \sigma[C][\mathrm{id}_{1:n} \leftarrow v_{1:n}]\rangle \overset{t_1}{\mapsto} \sigma' \qquad \langle e, \sigma'\rangle \overset{t_2}{\mapsto} v}{\langle F, \sigma, (v_1, \ldots, v_n)\rangle \overset{t_1, t_2}{\longmapsto} \langle \sigma[C \leftarrow \sigma'[C]], v\rangle}\ \texttt{function}$$

Figure 16: Semantics for contract functions.

$$\frac{\langle L, \sigma\rangle \overset{t_1}{\Vdash\!\!\rightarrow} l \quad \langle e, \sigma\rangle \overset{t_2}{\mapsto} v \quad \langle \alpha, \sigma\rangle \overset{t_3}{\mapsto} a}{\langle L = e, \sigma\rangle \overset{t_1, t_2, t_3, v@a}{\longmapsto} \sigma[l \leftarrow v]}$$

$$\frac{\langle p, \sigma\rangle \overset{t_0}{\longmapsto} \mathrm{Proof}_\phi(R; v_{1:n}; v'_{1:m}) \quad \phi(v_1, \ldots, v'_m) = 1 \quad \langle e_1, \sigma\rangle \overset{t_1}{\mapsto} v_1 \quad \ldots \quad \langle e_n, \sigma\rangle \overset{t_n}{\mapsto} v_n}{\langle \texttt{verify}_\phi(p, e_1, \ldots e_n), \sigma\rangle \overset{t_0, t_1, \ldots, t_n, 1@\texttt{all}}{\longmapsto} \sigma}$$

Figure 17: Semantics for selected statements. Typed by Fig. 12, where α is the privacy type of L.

$$\frac{C[f] = F \quad \langle F, \sigma[\texttt{me} \leftarrow a], (v_1, \ldots, v_n)\rangle \overset{t}{\mapsto} \langle \sigma', v\rangle}{\langle \mathrm{Tx}^{(a)}_{C.f}(v_1, \ldots, v_n), \sigma\rangle \overset{C@\texttt{all}, f@\texttt{all}, a@\texttt{all}, v_1@\alpha_1, \ldots, v_n@\alpha_n, t}{\Longrightarrow} \langle \sigma'[C], v\rangle}\ \texttt{T-ok}$$

$$\frac{C[f] = F \quad \langle F, \sigma[\texttt{me} \leftarrow a], (v_1, \ldots, v_n)\rangle \overset{t}{\mapsto} \texttt{fail}}{\langle \mathrm{Tx}^{(a)}_{C.f}(v_1, \ldots, v_n), \sigma\rangle \overset{C@\texttt{all}, f@\texttt{all}, a@\texttt{all}, v_1@\alpha_1, \ldots, v_n@\alpha_n, t, \mathrm{rollback}@\texttt{all}}{\Longrightarrow} \langle \sigma, \texttt{fail}\rangle}\ \texttt{T-fail}$$

Figure 18: Semantics for transactions. Here, the privacy level α_i is **all** if the i-th argument is public, and a otherwise.

repetitions by remembering previously simulated values for each runtime location.

We now describe how Sim extends $\overline{t}_2$ when following T in C step by step. All entries in $\overline{t}$ are public and, to avoid notational clutter, we will omit the privacy level @**all** from simulated trace entries in the remainder of the proof.

Start (transaction): The start of $\overline{t}$ is simulated by copying the beginning of t^*, where arguments private to honest accounts (which are hidden in t^*) are replaced by simulations using S1, and arguments private to dishonest accounts (whose plaintext is available in t^*) are replaced by simulations using S2. Transformation may have introduced three kinds of additional function arguments in $\overline{C}$, which are simulated as follows. *(i) Private arguments:* Similarly as the other private arguments, they are simulated using S1 and S2. *(ii) Public arguments:* Such an argument may only have been introduced by transformation as a result of transforming a declassification (see Fig. 5e). Hence, its plaintext is always contained in trace t^*, namely in the part where the declassification is evaluated, and can be copied by Sim. *(iii) NIZK proof:* Since the caller is honest, the proof is valid. The public arguments of the proof are available at the end of t^* where the arguments for **verify** (which are public by the type system) are evaluated. Knowing the public arguments, Sim uses S4 to simulate the NIZK proof.

Variable declarations, skip statements: Trivial to simulate.

Private Expressions: Private composite expressions are substituted by encrypted function arguments v_i during transformation. If the expression is private to an honest account in C, accessing v_i in $\overline{C}$ is simulated by S1. Otherwise, its plaintext value is available in t^* and accessing v_i can be simulated by S2. The only non-composite private expressions are private locations. Reading from these is simulated analogously, however, they may also contain encryptions generated by dishonest accounts, which are simulated by S3. Further, reading mapping entries involves resolving (public) mapping keys, which is simulated as described next.

Public Expressions: Evaluation of public expressions is simulated by copying the corresponding parts of t^* and simulating evaluation of any (potentially private, due to declassification) subexpressions. Resolving any mapping keys is simulated recursively.

Assignments: First, evaluating the runtime location of the assignment's left-hand side is simulated by copying the respective parts from t^* (runtime locations are always public in C due to the type system) and simulating evaluations of any mapping keys using expression simulation described before. Then, evaluation of the right-hand side expression is simulated as described before.

Require: The evaluation of the condition expression is simulated as described before. Because T is assumed to not throw an exception, it passes all require statements.

While: Since while-loops are enforced to be fully public by the type system, the corresponding part in t^* does not contain any hidden values. Transformation does not change loops and Sim simulates them by copying the corresponding parts in t^*.

If-then-else: The evaluation of the condition is simulated as described before. Because the condition is enforced to be public by the type system, we can determine the branch which is being executed by inspecting t^* and simulate that branch as described above.

End (proof verification): The transaction $\overline{T}$ ends with verifying the NIZK proof. The trace of evaluating the proof and all other arguments to $\texttt{verify}_\phi$ is simulated by the expression simulation described above. Further, Sim emits the trace entry 1@**all** to signify successful verification (because the caller is honest, the $\texttt{verify}_\phi$ statement is guaranteed to not throw an exception in $\overline{T}$). Note that no trace needs to be simulated for evaluating the proof circuit ϕ during verification in $\overline{T}$ (see Fig. 17).

D.2 Privacy for Dishonest Callers.

Let $\mathcal{A}$ be an arbitrary attacker and consider an arbitrary transaction T' issued on $\overline{C}$ by the attacker such that $\langle T', \overline{\sigma}\rangle \overset{\overline{t}}{\Rightarrow} \langle \overline{\sigma}', \overline{v}\rangle$ for some $\overline{t}, \overline{\sigma}', \overline{v}$. The trace $\overline{t}$ can easily be simulated by the attacker by *locally* running the execution steps of T' on $\overline{C}$ and initial state $\overline{\sigma}$, because no additional input by any honest account is required.