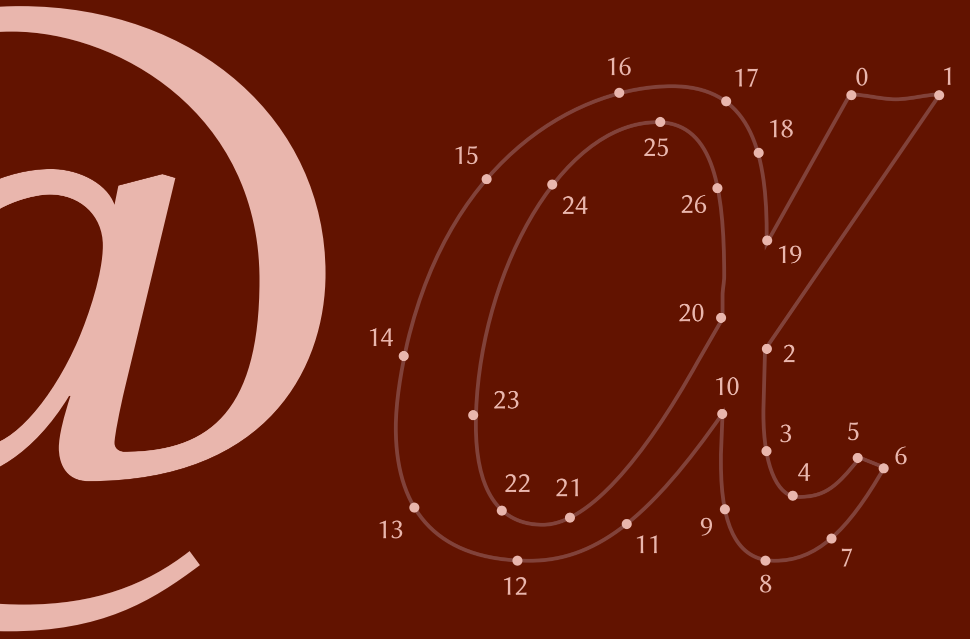
A Programming Language Approach to Smart Contract Privacy

Samuel Lutz Steffen



Diss. ETH No. 28962

DISS. ETH NO. 28962

A Programming Language Approach to Smart Contract Privacy

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

SAMUEL LUTZ STEFFEN
MSc CS, ETH Zurich

born on 7 August 1994

accepted on the recommendation of

Prof. Martin Vechev (ETH Zurich)
Prof. Ilya Sergey (National University of Singapore)
Prof. Elaine Shi (Carnegie Mellon University)

2022

In memory of Daniel Yu

ABSTRACT

In distributed ledgers (often called blockchains), a globally distributed state is updated by a history of irrevocable transactions. Modern blockchains allow programming these updates with custom logic using so-called smart contracts, which enables realizing decentralized applications without requiring a trusted third party. Typically, the data stored and processed on programmable blockchains is public, which prevents applications handling sensitive data from being ported to smart contracts.

In this thesis, we investigate how to ensure privacy for general smart contracts. While many works on private cryptocurrency transfers exist, the few proposals targeting general smart contracts suffer from various limitations and often require developers to instantiate advanced cryptographic primitives. In contrast, we adopt a programming language approach and design three systems usable by developers without cryptographic expertise.

First, we introduce the zkay language and compiler, which hide the data involved in smart contracts using encryption and non-interactive zero-knowledge (NIZK) proofs. The zkay language features a privacy type system allowing developers to express data ownership and preventing implicit information leaks. Our compiler automatically compiles zkay contracts to contracts executable on the popular Ethereum blockchain.

In our second system ZeeStar, we extend zkay to support computations on unknown private data—an essential feature required to implement important applications such as confidential payments. To this end, we modify zkay's type system and extend its compiler to instantiate additively homomorphic encryption.

Third, we explore how to not only hide the data but also the parties involved in a transaction. Specifically, we introduce the Zapper system, which hides the accessed objects and the identities of its users using a combination of Merkle hash trees, key-private encryption, and NIZK proofs. Zapper contracts are compiled to a custom assembly language, which is subject to an access control mechanism and executed on a NIZK processor.

For each system, we provide a proof demonstrating that it respects a well-defined notion of privacy. We implement all systems, relying on advanced techniques including elliptic curve embedding to achieve practical performance when combining cryptographic primitives. Finally, we demonstrate the systems' versatility and efficiency on a variety of example contracts.

ZUSAMMENFASSUNG

Distributed-Ledger-Technologien (häufig Blockchains genannt) aktualisieren einen global verteilten Zustand durch eine Abfolge unwiderruflicher Transaktionen. Moderne Blockchains erlauben es, diese Transaktionen mittels sogenannten Smart Contracts individuell zu programmieren. Dies ermöglicht die Realisierung dezentralisierter Anwendungen ohne eine vertrauenswürdige Drittperson. Typischerweise sind die auf programmierbaren Blockchains gespeicherten und verarbeiteten Daten öffentlich. Für Anwendungen welche sensible Daten bearbeiten wird dadurch jedoch ein Portieren auf Smart Contracts verhindert.

Diese Arbeit präsentiert neue Systeme für den Erhalt der Vertraulichkeit in Smart Contracts. Obwohl viele Publikationen zu vertraulichen Überweisungen von Kryptowährungen existieren, gibt es nur wenige Vorschläge, welche allgemeine Smart Contracts unterstützen. Leider unterliegen diese Systeme verschiedenen Einschränkungen und verlangen von Entwicklern häufig das manuelle Instanziieren kryptografischer Bausteine. Diese Arbeit verfolgt im Gegensatz dazu einen Ansatz basierend auf Programmiersprachen und präsentiert drei Systeme, welche von Entwicklern ohne kryptographische Expertise angewendet werden können

Als Erstes werden die Zkay-Sprache und -Compiler entworfen. Diese verstecken die in Smart Contracts verarbeiteten Daten mittels Verschlüsselung und nicht-interaktiven Zero-Knowledge-Beweisen. Basierend auf einem Vertraulichkeits-Typensystem erlaubt Zkay den Entwicklern das Ausdrücken von Datenbesitz und das Verhindern ungewollter Datenlecks. Der Zkay-Compiler transformiert Smart Contracts in der Zkay-Sprache zu Smart Contracts für die beliebte Ethereum Blockchain.

Zweitens wird Zkay durch das ZeeStar-System um die Möglichkeit zur Berechnung auf unbekannten Daten erweitert. Diese Funktionalität ist essentiell um wichtige Anwendungen wie vertrauliche Zahlungen zu realisieren. Das Zkay-Typensystem und der Zkay-Compiler werden so erweitert, dass diese zusätzlich additiv-homomorphe Verschlüsselung instanziieren.

Als Drittes wird untersucht, wie zusätzlich zur Vertraulichkeit der Daten auch die Vertraulichkeit der involvierten Parteien erhalten werden kann. Das resultierende Zapper-System versteckt die aufgerufenen Objekte und die Identitäten seiner Benutzer mittels Merkle-Bäumen, Key-Private-Verschlüsselung und nicht-interaktiven Zero-Knowledge-Beweisen. Zapper-

Programme werden in eine spezielle Assembler-Sprache transformiert, welche einer Zugriffskontrolle unterliegt und auf einem Zero-Knowledge-Prozessor ausgeführt wird.

Für jedes System wird ein mathematischer Beweis präsentiert, welcher zeigt, dass eine wohldefinierte Form von Vertraulichkeit respektiert wird. Alle Systeme werden implementiert, wobei zur Leistungserhaltung bei der Kombination kryptographischer Bausteine fortschrittliche Techniken wie das Einbetten elliptischer Kurven angewendet werden. Die Vielseitigkeit und Effizienz der Systeme werden schliesslich an verschiedenen Beispielanwendungen demonstriert.

PUBLICATIONS

This thesis is based on the following publications:

- **Samuel Steffen**, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. "zkay: Specifying and Enforcing Data Privacy in Smart Contracts." In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019. [1]
- **Samuel Steffen**, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. "ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zero-knowledge Proofs." In: *2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2022. [2]
- **Samuel Steffen**, Benjamin Bichsel, and Martin Vechev. "Zapper: Smart Contracts with Data and Identity Privacy." In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2022. [3]

  ⋆ **Distinguished Paper Award**

The following publications were part of my doctoral research but present results outside the scope of the material covered by this thesis:

- Jan Eberhardt, **Samuel Steffen**, Veselin Raychev, and Martin Vechev. "Unsupervised learning of API aliasing specifications." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2019. [4]
- Timon Gehr, **Samuel Steffen**, and Martin Vechev. "λPSI: exact inference for higher-order probabilistic programs." In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2020. [5]
- **Samuel Steffen**, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. "Probabilistic Verification of Network Configurations." In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. ACM, 2020. [6]

  ⋆ **Best Student Paper Award**

- Benjamin Bichsel, **Samuel Steffen**, Ilija Bogunovic, and Martin Vechev. "DP-Sniper: Black-Box Discovery of Differential Privacy Violations using Classifiers." In: *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021. [7]
- Anouk Paradis, Benjamin Bichsel, **Samuel Steffen**, and Martin Vechev. "Unqomp: Synthesizing Uncomputation in Quantum Circuits." In: *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2021. [8]
- Nikola Jovanović, Marc Fischer, **Samuel Steffen**, and Martin Vechev. "Private and Reliable Neural Network Inference." In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2022. [9]

ACKNOWLEDGEMENTS

Many people have supported me throughout my doctoral studies. First and foremost, I'd like to thank my advisor Prof. Martin Vechev for all his guidance. Your inputs were extremely valuable: you taught me how to ask the right questions, how to look for good answers, how to present complex concepts, and—more generally—how to do research. The inspiring, collaborative, productive, practically oriented and excitement-driven group atmosphere you curated was fundamental to my success as a doctoral student. Allowing me to take significant responsibility for teaching furthered fostered my personal growth. I'm also very grateful to my second advisor Prof. Laurent Vanbever for his support, particularly during the early stages of my studies. Further, I'd like to thank Prof. Ilya Sergey and Prof. Elaine Shi for agreeing to serve on my thesis committee.

I also owe my thanks to all my co-authors. Throughout my research, I was able to collaborate with many truly inspiring people: Roger Baumgartner, Benjamin Bichsel, Ilija Bogunovic, Jan Eberhardt, Marc Fischer, Timon Gehr, Mario Gersbach, Nikola Jovanović, Noa Melchior, Anouk Paradis, Veselin Raychev, and Petar Tsankov. The countless discussions over the whiteboard really pushed my research forward. A special thanks goes to Benjamin for his continuous support throughout many research projects, the long discussions about research and beyond, and all the fun times. Further, I owe most of my writing skills to the close collaboration with Petar during the early stages of my studies. I also thank Petar for introducing me to the SRI Lab and encouraging me to pursue a doctoral degree in the first place.

I had the pleasure to supervise many excellent Master students and interns. A special thanks goes to Nick Baumann for extending the original zkay implementation to a more practical system.

I am very grateful to all current and former members of the SRI Lab, who helped making this lab an extraordinary working environment. I always enjoyed our discussions over lunch and your second opinions. In particular, I'd like to thank Prof. Dana Drachsler-Cohen for sharing her views on research and for helping with all the early-stage research projects we were unfortunately never able to wrap up. Also, a special thanks goes to Maximilian Baader for all his advice on topics outside research, and for sharing his great sense of humor.

I was very happy to experience the excellent administrative support in the SRI Lab and the department. Thanks to Mirella Rutz and Fiorella Meyer for making everything run so smoothly. I am also grateful to Denise Spicher and the whole studies administration for supporting me both as a teacher and student.

Last but not least, I'd like to thank my family and my friends for their continuous mental support. This work would not have been possible without you.

CONTENTS

1 INTRODUCTION

Modern distributed ledgers or *blockchains* allow decentralized but trusted execution of programs (so-called *smart contracts*) without relying on a trusted third party. The key idea is to automatically enforce execution of program logic that has been previously agreed upon by the involved parties. Due to their versatility, smart contracts have gained significant popularity in recent years. For instance, smart contracts are used to implement custom "tokens" traded on the blockchain [10], to enable decentralized exchanges of assets [11], or even to conduct lotteries [12]. Also beyond financial applications, many real-world processes such as voting schemes [13], the collection of medical data [14], and power consumption measurements [15] are being ported to smart contracts.

The most popular blockchains are permissionless, allowing any user to join the network without explicit permission from a governing entity. The prevalent permissionless blockchain supporting smart contracts is Ethereum [16]. In Ethereum, developers can use the Solidity programming language to implement custom high-level application logic. In Fig. 1.1, we show a basic example contract modeling a custom coin. The contract uses the `bal` mapping to store the number of coins owned by each user (identified by their `address`). It provides a `transfer` function to transfer `val` coins to a recipient `to`, where the user calling the function is identified by its address `msg.sender`. The function checks whether the sender of the coin has sufficient balance, and then updates the `bal` mapping.

```
contract Coin {
  mapping(address => uint) bal;
  function transfer(uint val, address to) public {
    require(val <= bal[msg.sender]);
    bal[msg.sender] = bal[msg.sender] - val;
    bal[to] = bal[to] + val;
  }
}
```

Figure 1.1: A basic Solidity smart contract modeling a (non-private) coin.

By design, all data stored and processed on permissionless blockchains is public. In particular, function calls initiated by users (so-called *transactions*) are processed by the blockchain's *miner* nodes, which requires the transactions' operations and data to be made available to all miners. In the example of Fig. 1.1, this means that any miner can see the balances of all accounts, the number of coins transferred during a transfer, and the sender and recipient accounts. In practice, this can be undesirable, as individuals may be linked to their purchases and trading partners.

While exemplified in the context of a coin, the lack of privacy on permissionless blockchains is especially problematic for applications that handle sensitive information such as health data [14] or voting ballots [13]. In fact, there are many more applications which would benefit from a smart contract system that ensures privacy. For example, such a system would allow realizing private auctions, untraceable digital train tickets, or even a double-blind academic peer review system on a permissionless blockchain. [1]

The current lack of privacy on permissionless blockchains gives rise to the following key research question:

How can we ensure privacy for general smart contracts on permissionless blockchains?

In this thesis, we address the above question from a programming languages perspective by designing, analyzing, and implementing multiple systems for private smart contracts.

1.1 RELATED WORK

Before introducing the contributions of this thesis, we next provide an overview of previous and concurrent work addressing similar research questions. In later chapters, we will discuss further related works and compare these to the systems presented in this thesis on a technical level.

PRIVACY FOR PAYMENTS A long line of work brings privacy to cryptocurrency transactions. Most of the earlier works hide the relationship between transactions by collecting and "mixing" pending transactions. Systems employing such mixers include Dash [17], MixCoin [18], CoinJoin [19], CoinShuffle [20], and CoinParty [21]. Unfortunately, these systems only provide relatively weak privacy guarantees [22].

1 In Chapter 4, we will actually realize these applications.

Other works rely on advanced cryptographic primitives such as non-interactive zero-knowledge (NIZK) proofs. For instance, the popular Monero [23] blockchain leverages ring signatures and range proofs to hide both the transaction amounts and the involved parties. Similarly, Zerocoin [24] relies on signatures of knowledge and RSA accumulators. Its successor Zerocash [25] (commercially deployed as Zcash [26] and ported to Ethereum as ZETH [27]) leverages zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) to provide strong privacy guarantees. Similarly, BlockMaze [28] is based on zk-SNARKs. Finally, some works combine NIZK proofs and homomorphic encryption or commitments, including Zether [29, 30] and Quisquis [31].

Unfortunately, all these works focus on privacy for payments and do not support general smart contracts.

PRIVACY FOR SMART CONTRACTS Various works propose smart contract systems providing some notion of privacy.

Some of the existing systems trade privacy for additional trust assumptions. For example, Hawk [32] and Arbitrum [33] rely on trusted managers. Similarly, Ekiden [34] and FastKitten [35] leverage trusted hardware. While convenient, these additional assumptions arguably undermine the weak trust assumptions of permissionless blockchains.

The trust assumption of Hawk is weakened in zkHawk [36] and V-zkHawk [37]. However, these systems require interactive parties—a fundamental limitation in the context of permissionless blockchains, where parties are typically communicating asynchronously.

The recent SmartFHE [38] and ZEXE [39] systems provide strong privacy guarantees for general smart contracts with weak trust assumptions. However, implementing smart contracts for these systems requires manually instantiating a fully-homomorphic encryption scheme or zk-SNARKs, respectively, inhibiting their usage by developers which are not cryptographic experts.

1.2 THIS WORK

In this thesis, we aim to address the limitations of previous work. In particular, we investigate how to bring privacy to general smart contracts without introducing unnecessary additional trust assumptions, while providing a simple interface to developers which abstracts all cryptographic components.

FEATURES OF PRIVATE SMART CONTRACTS Before discussing our approach, we should first elaborate on what it means to provide "privacy" for smart contracts. In fact, there are multiple dimensions of privacy which are relevant to our context. A basic notion of privacy would just hide the memory contents stored and processed on the blockchain. In our example in Fig. 1.1, this means that the individual balances stored in the `bal` mapping, and the arguments `val` and `to` of the function `transfer` are hidden from third parties (for example, by encrypting them). This notion, which we call *data privacy* in this thesis, is very useful as it hides the contents of accessed memory. However, it does not necessarily hide *which* memory locations are accessed.

This gives rise to a second dimension of privacy, determining whether the accessed memory locations are hidden. In the example of Fig. 1.1, not hiding these would leak the sender and recipient addresses of the transfer as these addresses determine the indices of `bal` to be accessed.

A third dimension is concerned with whether the party creating a transaction can remain anonymous. In Fig. 1.1, if the transaction needs to be signed by the sender, then the signature may reveal the sender's identity (even if the values of all variables and the accessed memory locations are hidden). We say a smart contract system satisfies *identity privacy* if it is able to hide the identities of its involved parties. Typically, this requires both hiding the accessed memory locations and hiding the transaction creator. A system providing identity privacy is well-suited to implement a fully private coin.

Independently of these dimensions, an additional concern is whether a private smart contract system prohibits accidentally or implicitly leaking private information to other parties. For instance, in Fig. 1.1, the balance of the recipient is increased by `val`. If `val` is considered private information belonging to the sender, this update leaks private information to the recipient. While this amount of leakage is fundamentally necessary in our example (the recipient will always have to learn how many coins it receives), a good private smart contract system would prevent executing such a code without an explicit declaration by the programmer accepting the leak.

CHALLENGES Designing a smart contract system with all of the above privacy features is challenging. First, providing strong privacy guarantees for payments only is already difficult per se, as illustrated by the long line of work on this matter and multiple demonstrations of successful attacks [25, 40]. Extending existing systems with programmability is highly non-trivial, as these are typically tailored to payments.

Second, instantiating advanced cryptographic primitives such as zk-SNARKs or homomorphic encryption in a private smart contract system is challenging, as these primitives (i) typically expose a rather low-level interface; (ii) are not expressive enough on their own but must be instantiated in combination with other primitives; (iii) provide non-trivial guarantees, in particular when instantiated in combination; and (iv) often lead to prohibitively low performance if instantiated naively.

Further, preventing implicit information leaks requires a concept of data ownership and means to express explicit leakage, as well as a (static or dynamic) analysis of ownership to detect implicit leaks.

APPROACH To address the above challenges, we employ core techniques from the area of programming languages and use these to automatically instantiate advanced cryptographic primitives.

In particular, we leverage type systems, static program analysis, and compilation in order to transform high-level and intuitive privacy specifications into low-level cryptographic primitives. As core cryptographic primitives, we use zk-SNARKs, (partially) homomorphic encryption, and Merkle hash trees. An introduction to these primitives is provided in §1.3.

A main goal of this thesis is to provide an intuitive interface to developers. In particular, using our systems should not require any cryptographic expertise, but only familiarity with existing non-private smart contract programming languages such as Solidity.

DESIGNED SYSTEMS While exploring answers to the key research question stated on page 2, we have designed and implemented three systems for private smart contracts, presented in Chapters 2–4. We provide an overview of these systems in Tab. 1.1. The systems have different privacy features according to the dimensions discussed above (first four rows in Tab. 1.1), and come with different levels of expressivity and blockchain support (last two rows). Next, we briefly introduce these systems.

ZKAY In Chapter 2, we introduce the zkay system. Being the first system of its kind, zkay only provides data privacy: it allows developers to hide the memory contents involved in their smart contracts, but not the accessed memory locations or the transaction creator.

On a technical level, this chapter introduces the zkay programming language, which allows developers to specify their desired privacy

TABLE 1.1: Overview of the three systems presented in this thesis.

	zkay Chapter 2	**ZeeStar** Chapter 3	**Zapper** Chapter 4
hides memory contents	●	●	●
hides memory locations			●
hides transaction creator			●
prevents implicit leaks	●	●	
can compute on unknown data		●	
supported blockchain	Ethereum	Ethereum	custom ledger

notion using privacy annotations. In particular, the zkay language extends Solidity by *privacy types* indicating the owner of private values and expressions to explicitly reveal values to other parties. Using its privacy type system, zkay then prevents implicit leaks. Contracts written in the zkay programming language are automatically compiled to contracts executable on a permissionless blockchain, which are realized using zk-SNARKs and asymmetric encryption. Our implementation of zkay allows the resulting contracts to be deployed on Ethereum and natively interact with the Ethereum ecosystem.

A fundamental limitation of zkay is its inability to compute on unknown data. For instance, a private variant of the coin in Fig. 1.1 cannot be readily implemented in zkay without restructuring, as the used cryptographic primitives do not allow the sender to update the private balance `bal[to]` owned by the recipient.

ZEESTAR In order to address the expressivity limitations of zkay, we extend zkay to support restricted modifications of unknown data in Chapter 3. The resulting system, called ZeeStar, supersedes zkay as it provides the same privacy features but allows implementing a wider variety of use cases on Ethereum. For example, ZeeStar can be readily used to implement a private variant of the coin in Fig. 1.1, where the balances and transferred value are private. Still, ZeeStar does not provide identity privacy as accessed memory locations are not hidden and the transaction creator signs any transaction using its own key.

Technically, ZeeStar integrates homomorphic encryption as an additional cryptographic primitive into its compilation pipeline, allowing parties to modify unknown values encrypted for others. Instantiating such an encryption scheme together with zk-SNARKs is challenging:

these primitives must be instantiated in close combination and great care must be taken in order to maintain the desired correctness and privacy guarantees, as well as to achieve acceptable performance. On the front end, we modify the privacy type system of zkay to be aware of homomorphic operations, while only requiring minimal changes to the privacy annotations of zkay.

ZAPPER The previously introduced zkay and ZeeStar systems are designed for a blockchain featuring an account-based execution model (such as Ethereum). As this execution model is not well-suited to provide identity privacy, most existing systems for private payments follow an unspent transaction output (UTXO) model. In Chapter 4, we use ideas of these systems to design the novel private smart contract system Zapper. Zapper's execution model resembles the UTXO style and its design is hence significantly different to that of zkay and ZeeStar. Instead of having a static privacy type system with fine-grained ownership annotations, Zapper assigns a dynamic owner address to each object as a whole. While this system currently does not support prevention of implicit leaks, it provides identity privacy by hiding the transaction creator and the accessed memory locations.

Conceptually, Zapper can be thought of as an extension of Zerocash [25] or Zcash [26], allowing coins to be programmed with custom logic. On a technical level, Zapper combines Merkle hash trees, zk-SNARKs, and key-private asymmetric encryption.

While Zapper has privacy features incomparable to the other two systems, it allows readily implementing a fully private coin, a double-blind peer review system, and further interesting applications. On the other hand, Zapper is not designed to work with the Ethereum blockchain, and does not allow computation on unknown data.

1.3 CRYPTOGRAPHY BACKGROUND

This thesis aims to be accessible to a general computer science audience. As it relies on advanced cryptographic primitives, we next provide an introduction to these primitives.

1.3.1 *Non-interactive Zero-knowledge Proofs*

A non-interactive zero-knowledge (NIZK) proof [41, 42] allows a prover to demonstrate to a verifier that she knows a secret, without revealing that secret. More precisely, she can prove knowledge of a secret witness w satisfying a predicate $\phi(x; w)$ for some public value x, without revealing anything else about w other than the fact that $\phi(x; w)$ holds. In this thesis, we call ϕ the *proof circuit*, w the *private* (or *secret*) *input*, and x the *public input*.

For example, for a cyclic group G with generator g and $h \in G$, one can prove knowledge of the discrete logarithm z of h for base g using the proof circuit $\phi(h; z)$ satisfied iff $g^z = h$.

Zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) [43, 44, 45] are generic NIZK proof constructions supporting any arithmetic circuit ϕ and featuring constant-cost proof verification in the size of ϕ (plus a typically negligible linear cost in the size of x). Due to their low verification costs, zk-SNARKs are frequently used on the Ethereum blockchain [46].

1.3.2 *Additively Homomorphic Encryption*

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

$$\text{Enc}(x, pk_\alpha, r) \oplus \text{Enc}(y, pk_\alpha, r') = \text{Enc}(x + y, pk_\alpha, r'') \tag{1.1}$$

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

For example, the Paillier encryption scheme [47] is additively homomorphic in $\mathbb{Z}_n$ (that is, $+$ in Eq. (1.1) is addition modulo n) for an RSA modulus n, and exponential ElGamal encryption [48] over a group G is additively homomorphic in $\mathbb{Z}_{|G|}$, where $|G|$ is the order of G (see App. A.4).

1.3.3 *Merkle Hash Trees*

A Merkle hash tree [49] is a cryptographic data structure maintaining a list of elements, allowing efficient summarization of the list and containment checks of elements. In a Merkle hash tree, elements are stored as leaves of a binary tree whose root contains a root hash r summarizing its contents.

More formally, for a collision-resistant hash function H, the value of a leaf holding an element v is $H(v)$, where empty elements are represented by $v = 0$. The value of an inner node (including the root node) is $H(v_1 \parallel v_2)$, where v_1 and v_2 are the values of its two children. A Merkle hash tree of height l hence maintains 2^l elements.

Elements can be inserted into a Merkle hash tree T by replacing an empty leaf and recomputing the root hash r. Further, one can certify the containment of a leaf in T using a *Merkle certificate* [50, §2.1.1], which contains the list of sibling values along the root-to-leaf path in T.

1.4 thesis contributions

This thesis makes the following contributions.

- In Chapter 2, we introduce the zkay programming language for writing private smart contracts. Its privacy type system allows tracking ownership of data and detecting implicit information leaks. We present an automatic compilation of zkay contracts into contracts executable on a permissionless account-based blockchain such as Ethereum. The compilation automatically instantiates asymmetric encryption and zk-SNARKs. We present a formal definition of data privacy for zkay along with a simulation-based proof in the symbolic model showing that our compilation respects privacy. We implemented our approach for Ethereum[2] and evaluated it on 10 example contracts. We demonstrate that zkay can express interesting applications, offers significant advantages over manual instantiation of NIZK proofs, and results in moderate costs.

- In Chapter 3, we present an extension of zkay to improve its expressivity and realize this extension in the novel ZeeStar system. In particular, we extend zkay's privacy type system to allow addition-based modifications of values owned by other parties, and show how zkay's compilation pipeline can be extended by additively homomorphic

2 Publicly available at `https://github.com/eth-sri/zkay/tree/ccs2019`

encryption in order to realize such modifications. We further present how we can add support for private multiplication of unknown values, and mixing non-homomorphic and homomorphic encryption schemes. We present a simulation-based security proof in the computational model demonstrating that the resulting system respects data privacy. We implemented ZeeStar for Ethereum[3] and demonstrate in our evaluation of 12 example contracts that ZeeStar is expressive and practical.

- In Chapter 4, we introduce the private smart contract system Zapper, which provides data and identity privacy for rich smart contracts expressed in a Python-embedded frontend. We present a compilation of such contracts to a custom assembly language, as well as an access control system ensuring the data held by smart contracts is protected from untrusted code. Further, we present a cryptographic construction inspired by previous work to maintain and efficiently update the system state while satisfying key security properties. We present a simulation-based security proof in the computational model demonstrating that Zapper provides data and identity privacy. We implemented Zapper[4] on top of a custom ledger and demonstrate on 12 example classes that Zapper is efficient.

3 Publicly available at `https://github.com/eth-sri/zkay/tree/sp2022`
4 Publicly available at `https://github.com/eth-sri/zapper`

2 A LANGUAGE AND COMPILER FOR SMART CONTRACTS WITH DATA PRIVACY

In this chapter, we make the first steps to bring privacy to permissionless blockchains. Specifically, we present a base system called zkay, which provides data privacy on Ethereum without yet focusing on expressivity or identity privacy. In particular, this system does not allow computation on unknown data—a restriction we will relieve in Chapter 3.

In contrast to all existing systems for smart contract privacy (see §2.10), zkay follows a programming-language based approach, allowing developers to specify their desired privacy notion in an ergonomic manner directly in the input contracts, and compiling this specification to an equivalent "public" smart contract.

2.1 INTRODUCTION

As we already discussed in §1.1, multiple previous works have proposed systems for private smart contracts. Broadly, these can be categorized into two classes: The first class of systems introduces additional trust assumptions by relying on trusted managers or hardware (e.g., Hawk [32], Arbitrum [33], and Ekiden [34]). However, a fundamental property of permissionless blockchains is the relatively low level of trust required. Therefore, introducing these trusted entities is arguably undesirable. A second class of systems leverages cryptographic primitives to achieve privacy without undermining the trust model of permissionless blockchains. However, as we discuss in §2.10, the existing systems in this class unfortunately suffer from individual shortcomings.

THE PROMISE OF NIZK PROOFS A "folklore" approach to achieve data privacy on permissionless blockchains is to apply the following construction [51]: 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 (see §1.3.1) certifying that the encrypted values are correct with respect to f.

LIMITATIONS OF NIZK PROOFS Fortunately, practical NIZK proof constructions have been proposed [43, 52, 53, 54] and made available in Ethereum [46, 55]. However, while the above 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 [56]) supporting features—such as unbounded state and loops[1]—that cannot be captured by NIZK proof circuits. 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* (i.e., are performed on the blockchain).

(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. 2.1b (contract) and Fig. 2.1c (NIZK proof circuit); 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.

1 Relying on Ethereum's block gas limit to derive a bound on the number of loop iterations is not straightforward as this limit is dynamic and the amount of gas required may vary with each loop iteration.

(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.

ZKAY To address these challenges, we introduce zkay, a language cleanly separating the task of *specifying* logic and ownership of private data from the task of *realizing* this specification using NIZK proofs.

SPECIFICATION We show an example zkay contract in Fig. 2.1a. 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 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.

```
contract MedStats {
  final address hospital;
  uint@hospital count;
  mapping(address!x => bool@x) risk;

  constructor() {
    hospital = me;
    count = 0;
  }

  function record(address don, bool@me r) {
    require(hospital == me);
    risk[don] = reveal(r, don);
    count = count + (r ? 1 : 0);
  }

  function check(bool@me r) {
    require(reveal(r == risk[me], all));
} }
```

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

```
contract MedStats {
  final address hospital;
  bin count;
  mapping(address => bin) risk;

  constructor(bin v0, bin proof) {
    hospital = me;
    count = v0;
    verify_χ(proof, v0, pk(me));
  }

  function record(address don, bin r, bin v0, bin v1, bin proof) {
    require(hospital == me);
    risk[don] = v0;
    v2 = count;
    count = v1;
    verify_ϕ(proof, r, v0, v1, v2, pk(don), pk(me));
  }

  function check(bin r, bool v0, bin proof) {
    require(v0); verify_ψ(proof, r, risk[me], v0);
} }
```

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

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

(c) Proof circuit ϕ for function record.

FIGURE 2.1: Example illustrating the transformation of a zkay contract into an equivalent but fully public zkay contract.

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 [56] 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).

OUTLINE The remainder of this chapter is organized as follows.

- After giving an overview of zkay (§2.2), we introduce the zkay language for writing private smart contracts (§2.3) and provide its formal semantics (§2.4).
- In §2.5, we present a privacy-preserving transformation of zkay contracts into equivalent, fully public zkay contracts.
- Next, in §2.6 we provide a formal definition of privacy for zkay contracts along with a simulation-based proof that each transformed contract is private with respect to its specification contract.
- In §2.7–§2.8 we present an implementation and evaluation of our approach, followed by a discussion of possible extensions in §2.9.
- Finally, in §2.10–§2.11 we discuss related work and conclude the chapter.

2.2 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 smart contract executable on a permissionless blockchain.

EXAMPLE: MEDICAL STATISTICS Contract `MedStats` in Fig. 2.1a 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 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).

2.2.1 *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 τ@α (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 `m[`α`][`β`]` is private to α.

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

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

FIGURE 2.2: 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'.

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). The type system of zkay ensures that **`final`** variables remain constant. This prevents *ownership transfer*, which we disallow in zkay for simplicity. In §2.9, 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. The interpretation of privacy types will become more clear when we discuss the semantics of zkay (§2.4).

TYPE SYSTEM EXEMPLIFIED We now introduce zkay's type system and illustrate how it checks function `record`, addressing challenges C1–C4. We present three key rules of zkay's type system (illustrated shortly) in Fig. 2.2; further details on the type system will be given in §2.3.

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. 2.2) 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. 2.2) 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 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 [57]) 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. 2.2) 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.

2.2.2 *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 (§2.5). 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. 2.1b, 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 $\text{Enc}(\textbf{true}, R, pk_{0x01})$, where R is some randomness and 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. 2.1a) is fully public, we do not transform it (cp. Line 13, Fig. 2.1b).

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. 2.1b). To force the caller to provide the correct value for `v0`, we collect a correctness constraint in ϕ (see Fig. 2.1c, this represents the proof circuit 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 private inputs `R0` and `sk` of ϕ 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 secrets such that ϕ is satisfied together with the remaining arguments provided on-chain (see next). This proof is verified in Line 17 of Fig. 2.1b, 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 **verify**$_\phi$ can be realized on public blockchains using tools like ZoKrates [46].

Finally, we transform Line 14 (Fig. 2.1a) to Line 16 (Fig. 2.1b), 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. 2.1b, 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. 2.1b) so to avoid passing whole mappings to the verifier (see C1 in §2.1). This requires us to disallow encrypting mappings as a whole and enforce 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.

PRIVACY It is not clear a priori 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 this chapter, 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 (§2.6) 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 $\bar{t}$.

2.3 THE ZKAY LANGUAGE

We now discuss the zkay language and its privacy type system in more detail.

2.3.1 *Syntax*

Fig. 2.3 shows the syntax of zkay, which is inspired by Solidity [56]. In order to focus on key insights, zkay is deliberately kept simple.

The zkay language 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. 2.3). 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 §2.9.

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`**`(`τ_1 `=>` $\tau_2@\alpha_2$`)`) and *named* mappings of the form

`mapping``(`**`address`**`!`id `=>` $\tau@\alpha$`)`,

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 floating point numbers, structs, and arrays (conceptually, arrays are equivalent to **`mapping`**`(`**`uint`** `=>` $\tau@\alpha$`)`).

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

$$
\begin{array}{lll}
e & ::= c \mid \texttt{me} \mid L \mid \texttt{reveal}(e,\alpha) \mid \ominus e \mid e_1 \oplus e_2 \mid e_1 \,\texttt{?}\, e_2 \,\texttt{:}\, e_3 \mid \texttt{pk}(e) \mid \cdots & \text{(Expression)} \\
\tau & ::= \texttt{bool} \mid \texttt{uint} \mid \texttt{address} \mid \texttt{bin} \mid \texttt{mapping}(\tau_1 \texttt{ => } \tau_2@\alpha_2) \mid \texttt{mapping}(\texttt{address!}\mathrm{id} \texttt{ => } \tau@\alpha) & \text{(Data type)} \\
P & ::= \texttt{skip} \mid \tau@\alpha \ \mathrm{id} \mid L \texttt{ = } e \mid P_1; P_2 \mid \texttt{require}(e) \mid \texttt{if } e \ \{P_1\} \texttt{ else } \{P_2\} \mid \texttt{while } e \ \{P\} \mid \texttt{verify}_\phi(e_0, e_1, \ldots, e_n) & \text{(Statement)} \\
F & ::= \texttt{function } f(\tau_1@\alpha_1 \ \mathrm{id}_1, \ldots, \tau_n@\alpha_n \ \mathrm{id}_n) \texttt{ returns } \tau@\alpha \ \{P; \texttt{return } e;\} & \text{(Function)} \\
C & ::= \texttt{contract } \mathrm{id}\{(\texttt{final})?\ \tau_1@\alpha_1 \ \mathrm{id}_1; \ \ldots \ (\texttt{final})?\ \tau_n@\alpha_n \ \mathrm{id}_n; \ F_1 \ldots F_m\} & \text{(Contract)}
\end{array}
$$

FIGURE 2.3: Syntax of zkay, where f and 'id' are identifiers, c is a constant, and ϕ an arithmetic circuit. Native functions are highlighted .

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 $\texttt{verify}_{\phi}(e_0, e_1, \dots, e_n)$, 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 $v_1, \dots, v_m$ such that ϕ returns 1 when given n public arguments $e_1, \dots, e_n$ and secret arguments $v_1, \dots, v_m$. 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. 2.1a). Contracts (C) consist of contract field declarations and function declarations, where contract fields may be declared **final**.

2.3.2 *Privacy Type System*

As the typing rules to derive data types are standard, we only discuss privacy types here.

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 $\alpha, \alpha_1, \dots, \alpha_n \in \{\texttt{me}, \texttt{all}\}$. 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.

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 ensures that such expressions can only be used as right-hand sides of assignments (e.g., $\texttt{reveal}(10, x)$ for x of type **address@all**).

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

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

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

FIGURE 2.4: Typing rules for expressions.

Expression **`me`** has type **`address@all`**. The remaining typing rules for expressions are shown in Fig. 2.4. The rule for constants c indicates they are always public. Here, $[\![\tau]\!]$ denotes the set of values with type τ. For example, $[\![\mathtt{uint}]\!]$ denotes non-negative integers. We provide the formal definition of $[\![\cdot]\!]$ in App. A.2.

The next rule describes how $\mathtt{reveal}(e, \alpha)$ 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 $\alpha = \mathtt{all}$), or reclassifying information for some other owner $\alpha \notin \{\mathtt{me}, \mathtt{all}\}$. 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 $\alpha \in \{\mathtt{me}, \mathtt{all}\}$. 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 lightweight Abstract Interpretation [57] 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 **`address@all`**, **`me`**, or **`all`**. Although **`all`** is technically not an expression, we

$$\frac{x\colon \tau@\alpha \in \Gamma}{\Gamma \Vdash x\colon \tau@\alpha} \qquad \frac{\Gamma \Vdash L\colon \texttt{mapping}(\tau \texttt{ => } \tau'@\alpha)@\texttt{all} \quad \Gamma \vdash e\colon \tau@\texttt{all}}{\Gamma \Vdash L[e]\colon \tau'@\alpha}$$

$$\frac{\begin{array}{cc} \Gamma \Vdash L\colon \texttt{mapping}(\texttt{address!}\mathrm{id} \texttt{ => } \tau@\alpha)@\texttt{all} & \mathrm{id} \notin \Gamma \\ \Gamma \vdash e\colon \texttt{address@all} & (e = \mathrm{id}' \ \vee \ e = \texttt{me}) \end{array}}{\Gamma \Vdash L[e]\colon \tau@\alpha[\mathrm{id} \mapsto e]}$$

FIGURE 2.5: Typing rules for locations.

also assign it the type **address@all** for simplicity, meaning that all privacy types are of type **address@all**.

LOCATIONS Fig. 2.5 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 id$'$ or **me**.

STATEMENTS The rules for sequential composition and **skip** statements are standard. The condition of loops and if-then-else statements is enforced to be of type **bool@all** (to prevent implicitly leaking the condition's value). Further, the body and condition of loops cannot involve any private variables.

We show typing rules for the remaining statements in Fig. 2.5. 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. 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

$$\frac{\Gamma \vdash \alpha\colon \texttt{address@all} \quad \mathrm{id} \notin \Gamma}{\Gamma \xrightarrow{\tau@\alpha\ \mathrm{id}} \Gamma, \mathrm{id}\colon \tau@\alpha}\ \texttt{decl}$$

$$\frac{\Gamma \Vdash L\colon \tau@\alpha \quad \Gamma \vdash e\colon \tau@\alpha' \quad (\alpha = \alpha' \vee \alpha' = \texttt{all})}{\Gamma \xrightarrow{L=e} \Gamma} \qquad \frac{\Gamma \vdash e\colon \texttt{bool@all}}{\Gamma \xrightarrow{\texttt{require}(e)} \Gamma}$$

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

FIGURE 2.6: Typing rules for statements.

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. 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, $e_1 ?\ e_2 : e_3$ 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**.

2.4 SEMANTICS BY EXAMPLE

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

2.4.1 *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 (§2.6). 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 $\mathrm{Tx}^{(a)}_{C.f}(v_{1:n})$ 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** for a reserved failure value **fail**.

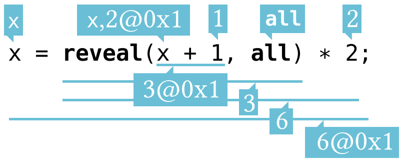
2.4.2 *Example Transaction*

We introduce the key aspects of zkay's semantics on an example (formal semantics for zkay is provided in App. A.2). Consider the contract in Fig. 2.7a 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):

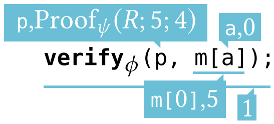
$$\phi(v_1; v_2) = 1 \quad \Longleftrightarrow \quad v_1 - 1 = v_2.$$

```
1 contract C {
2   mapping(uint => uint) m;
3   function f(uint a, uint@me x, bin p) {
4     x = reveal(x + 1, all) * 2;
5     verify_φ(p, m[a]);
6 } }
```

(A) Example contract C.



(B) Trace emitted by Line 4 of C.



(C) Trace emitted by Line 5 of C.

FIGURE 2.7: The semantics of zkay illustrated for an example transaction where address 0x1 calls $\texttt{f}(0,2,\text{Proof}_\psi(R;5;4))$ on C, assuming $\texttt{m[0]} = 5$.

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 $\text{Enc}(v, R, pk_a)$ 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 [58]. 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. 2.7a), 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. 2.7b 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

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

and the new state is

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

PROOF VERIFICATION Next, we describe how the proof `p` is verified in Line 5. Fig. 2.7c 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."

2.5 TRANSFORMATION

We now describe how to transform a zkay contract C to a fully public zkay contract $\overline{C}$ (§2.5.1–§2.5.3) while preserving privacy (discussed in §2.6). Because $\overline{C}$ has a different interface than C, we also discuss how to transform transactions T on C to transactions $\overline{T}$ on $\overline{C}$ (§2.5.4).

CORRECTNESS By construction, our transformation ensures correctness, as formalized in Thm. 2.1.

Theorem 2.1 (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, $\overline{t}$.*

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 $v' = \text{Enc}(v, R, pk_a)$ 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.

2.5.1 *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

```
function f(...) {
  $P$;
  return e;
}
```

$\Rightarrow$

```
function $\overline{\text{f}}$(..., $\mathcal{F}$, bin proof) {
  $T(P)$
  verify$_\phi$(proof, $\mathcal{P}$);
  return $T_{\text{e}}(\text{e})$;
}
```

for $\phi \colon (\mathcal{P}, \mathcal{S}) \to \{0,1\}$

(A) Transformation of a zkay function.

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

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

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

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

FIGURE 2.8: Overview of zkay transformations (part 1). We write $e@\alpha$ to indicate that e has privacy type α. The symbol v_i denotes a fresh variable.

$$T(P_1; P_2) ::= T(P_1); T(P_2) \tag{2.1}$$

$$T(L@\alpha = e@\alpha) ::= T_L(L) = T_{\mathsf{e}}(e) \qquad \text{(we use } T_L = T_{\mathsf{e}}\text{)} \tag{2.2}$$

$$T(L@\alpha = e@\mathbf{all}) ::= T_L(L) = out(e, \alpha) \qquad (\alpha \neq \mathbf{all}) \tag{2.3}$$

$$T(\mathbf{require}(e)) ::= \mathbf{require}(T_{\mathsf{e}}(e)) \tag{2.4}$$

$$T(\mathbf{while}\ e\ \{P\}) ::= \mathbf{while}\ e\ \{P\} \qquad (P \text{ is fully public}) \tag{2.5}$$

$$\begin{aligned} T(\mathbf{if}\ e\ \{P_1\}\ \mathbf{else}\ \{P_2\}) ::= {} & \mathbf{if}\ \ (T_{\mathsf{e}}(e))\ \{T(P_1, T_{\mathsf{e}}(e))\} \\ & \mathbf{else}\ \{T(P_2, T_{\mathsf{e}}(!e))\} \end{aligned} \tag{2.6}$$

(A) Transforming statements using T.

$$T_{\mathsf{e}}(c) ::= c \qquad \text{const } c \tag{2.7}$$

$$T_{\mathsf{e}}(\text{id}) ::= \text{id} \qquad \text{var id } (\star) \tag{2.8}$$

$$T_{\mathsf{e}}(L[e]) ::= T_L(L)[T_{\mathsf{e}}(e)] \qquad \text{mapping entry} \tag{2.9}$$

$$T_{\mathsf{e}}((e_1 + e_2)@\mathbf{all}) ::= T_{\mathsf{e}}(e_1) + T_{\mathsf{e}}(e_2) \qquad \text{native functions} \tag{2.10}$$

$$T_{\mathsf{e}}(\mathbf{reveal}(e, \alpha)) ::= out(e, \alpha) \qquad \text{(invariant: } e@\mathbf{me}\text{)} \tag{2.11}$$

$$T_{\mathsf{e}}(e@\alpha) ::= out(e, \alpha) \qquad \text{(invariant: } e@\mathbf{me}\text{)} \tag{2.12}$$

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

$$T_\phi(c) ::= c \qquad \text{const } c \tag{2.13}$$

$$T_\phi(L@\alpha) ::= in(L, \alpha) \qquad \text{(invariant: } \alpha \in \{\mathbf{all}, \mathbf{me}\}\text{)} \tag{2.14}$$

$$T_\phi(\mathbf{reveal}(e, \alpha)) ::= T_\phi(e) \tag{2.15}$$

$$T_\phi(e_1 + e_2) ::= T_\phi(e_1) + T_\phi(e_2) \qquad \text{native functions} \tag{2.16}$$

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

FIGURE 2.9: Overview of zkay transformations (part 2). We write $e@\alpha$ to indicate that e has privacy type α.

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.

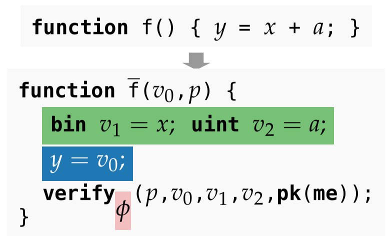
Figs. 2.8–2.9 show an overview of our transformation. To avoid notational clutter, the figures do not describe how to transform the types of private locations. Because these hold encrypted values in $\overline{C}$, our transformation changes their type to **`bin@all`**. An example is `count` in Line 3 of Fig. 2.1a, transformed to Line 3 of Fig. 2.1b.

TRANSFORMING FUNCTIONS A well-typed zkay contract C is transformed by transforming all its functions according to Fig. 2.8a. The function body and returned expression are transformed using statement transformation T (Fig. 2.9a) and expression transformation T_e (Fig. 2.9b), 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 [46] to instantiate $\mathbf{verify}_{\phi}$ for a given ϕ, hence the latter ultimately needs to be encoded in the DSL of ZoKrates. 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 $\mathbf{enc}(x, R, k)$ encrypts x using randomness R and public key k to yield the (symbolic) value $\text{Enc}(x, R, k)$ of data type **`bin`**, while $\mathbf{dec}_{\tau}(x, k)$ 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. 2.1c shows an example of a proof circuit encoding.

2.5.2 *Transformation Example*

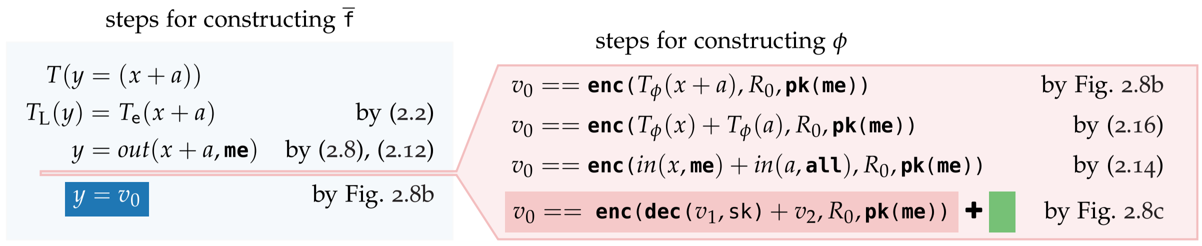
We now provide an intuition of the transformation defined in Figs. 2.8–2.9 using a concrete example. Particularly, we discuss how the function `f` shown in Fig. 2.10a is transformed step-by-step (Fig. 2.10b) to $\overline{\texttt{f}}$.



```
function f() { y = x + a; }
```

```
function f̄(v0,p) {
  bin v1 = x; uint v2 = a;
  y = v0;
  verify_φ(p,v0,v1,v2,pk(me));
}
```

(A) Transforming f to f̄.



steps for constructing f̄

$$T(y = (x + a))$$
$$T_L(y) = T_e(x + a) \quad \text{by (2.2)}$$
$$y = out(x + a, \texttt{me}) \quad \text{by (2.8), (2.12)}$$
$$y = v_0 \quad \text{by Fig. 2.8b}$$

steps for constructing ϕ

$$v_0 == \texttt{enc}(T_\phi(x + a), R_0, \texttt{pk(me)}) \quad \text{by Fig. 2.8b}$$
$$v_0 == \texttt{enc}(T_\phi(x) + T_\phi(a), R_0, \texttt{pk(me)}) \quad \text{by (2.16)}$$
$$v_0 == \texttt{enc}(in(x, \texttt{me}) + in(a, \texttt{all}), R_0, \texttt{pk(me)}) \quad \text{by (2.14)}$$
$$v_0 == \texttt{enc}(\texttt{dec}(v_1, \texttt{sk}) + v_2, R_0, \texttt{pk(me)}) + 🟩 \quad \text{by Fig. 2.8c}$$

(B) Step-by-step transformation of the assignment y@**me** $= x$@**me** $+ a$@**all** (emitted parts are highlighted).

FIGURE 2.10: 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.

TRANSFORMING STATEMENTS We begin by transforming the body of `f` using the statement transformation T formally defined in Fig. 2.9a. Fig. 2.10b (left, first two lines) shows how we apply rule (2.2).

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 $\overline{\mathtt{f}}$, 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. 2.9b). 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 (2.12). Because $x + a$ cannot be evaluated on-chain in $\overline{\mathtt{f}}$ 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 $\overline{\mathtt{f}}$ (i.e., moving $x + a$ out of ϕ). This is achieved using $out(x + a, \mathtt{me})$, 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. 2.9b (discussed in §2.5.3), we do not discuss it further. In our example, we apply rule (2.8).

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 $\overline{\mathtt{f}}$, e in C is equivalent to $T_{\mathsf{e}}(e)$ 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 $T_{\mathrm{L}}(L)$ in $\overline{C}$ evaluates to the same runtime location as L in C.

MOVING VALUES OUT OF THE PROOF CIRCUIT The transformation *out* (Fig. 2.8b) ensures that the correct encryption of $x + a$ is computed in ϕ and "moved out" to $\overline{\mathtt{f}}$. It performs two steps: First, it replaces $x + a$ by a fresh function argument v_0 (see also Fig. 2.10a) and thereby concludes the transformation in $\overline{\mathtt{f}}$ (see highlighted in Fig. 2.10). Then, it continues transformation inside the proof circuit ϕ (see right box in Fig. 2.10b). 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. 2.9c). Addition can directly be performed in ϕ and we apply rule (2.16). Next, we apply rule (2.14) 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 $\overline{\texttt{f}}$. To make the cleartext value of x available in ϕ, *in* (Fig. 2.8c) performs the following steps. First, it passes the current (encrypted) value of x from $\overline{\texttt{f}}$ 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 $\overline{\texttt{f}}$ (see highlighted in Fig. 2.10a, 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 $\overline{\texttt{f}}$ and no decryption is required.

The resulting constraint in ϕ (see highlighted in Fig. 2.10b, 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).

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. 2.1c)

2.5.3 *Additional Rules*

We now describe some additional transformation rules.

STATEMENTS Rule (2.3) 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 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 $T_{\mathsf{e}}(P, b)$ are only enforced if b is true by replacing them by $!b \mathbin{||} c$. To avoid clutter, we do not incorporate guard conditions in Fig. 2.8.

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 (2.8)). This is not required for T_{L}.

Rules (2.10) and (2.11) 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.

2.5.4 *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. 2.8b. 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.

2.6 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 (§2.6.1), and how C is executed in an ideal world (§2.6.2). 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 (§2.6.3).

At a high level, we prove a simulation-based indistinguishability notion of privacy. To support intuition and simplify the privacy proof, we employ a symbolic view with perfect cryptography in this chapter. However, in Chapter 3 we provide a more standard computational proof for an extension of zkay, thereby increasing confidence in the security of the system.

2.6.1 *Attacker Model*

We consider an active attacker interacting with a public blockchain as modeled by our semantics (§2.4). 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 [59], 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. [58]).

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 $R, R' \in \mathsf{R}_{\mathsf{hon}}$:

$$\mathrm{Enc}(m, R, pk_b) \sim_{\mathcal{A}} \mathrm{Enc}(m', R', pk_b)$$

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

$$\mathrm{Enc}(m, R, pk_a) \sim_{\mathcal{A}} \mathrm{Enc}(m, R', pk_a)$$

(iv) For any ϕ, $v_{1:n}$, $v'_{1:m}$ 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 §2.5, 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.

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 [60].

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

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 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.

2.6.2 *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 $\mathcal{A} = \{$**all**$, \text{0x0}\}$, it is $\mathrm{obs}_\mathcal{A}(t) = 1, \square, 3$.

2.6.3 *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

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 [42].

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. 2.1 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 §2.5.4). 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 2.1 (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 σ, $\overline{\sigma}$ as defined above the following holds: if $\langle T,\sigma\rangle \overset{t}{\Rightarrow} \langle\sigma',v\rangle$ 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 $\overline{t}$, $\overline{\sigma}'$, $\overline{v}$, with $\overline{T}$ being the transformation of T, then $\overline{t} \sim_{\mathcal{A}} \overline{t}_2$ for $\overline{t}_2 = Sim(\overline{\sigma}, obs_{\mathcal{A}}(t), C)$.*

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 2.2 (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*

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.

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 $\overline{t}_2 = Sim'(\overline{\sigma}, T', C)$.

PRIVACY THEOREM Thm. 2.2 states that our transformation respects privacy according to Def. 2.1 and Def. 2.2.

Theorem 2.2. *The contract $\overline{C}$ transformed from a well-typed contract C according to §2.5 is private w.r.t. C for honest and dishonest callers.*

2.6.4 *Privacy Proof*

We now provide a proof of Thm. 2.2. 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.

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. We consider privacy for dishonest and honest callers separately.

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. □

HONEST CALLERS We next construct the simulator Sim for honest caller transactions. Let $\mathcal{A}$ be an arbitrary attacker, let σ be a state in C that is equivalent to some $\overline{\sigma}$ in $\overline{C}$, and assume $\langle T, \sigma \rangle \overset{t}{\Rightarrow} \langle \sigma', v \rangle$ for some T, t, σ', $v \neq \texttt{fail}$ and $\langle \overline{T}, \overline{\sigma} \rangle \overset{\overline{t}}{\Rightarrow} \langle \overline{\sigma}', \overline{v} \rangle$ for some $\overline{t}$, $\overline{\sigma}'$, $\overline{v}$. Below, we show how Sim constructs a trace $\overline{t}_2$ indistinguishable from $\overline{t}$, given $\overline{\sigma}$, $t^* := \text{obs}_{\mathcal{A}}(t)$ 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 introducing four simulation procedures (S1–S4).

(S1) Encryptions created by honest for honest accounts: Sim simulates encryptions of the form $\text{Enc}(m, R, pk_a)$ for some m, $R \in \mathsf{R}_{\mathsf{hon}}$, and $a \notin \mathcal{A}$ by computing $\text{Enc}(0, R', pk_a)$ 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 §2.6.1.

(S2) Encryptions created by honest for dishonest accounts: Such encryptions have the form $\text{Enc}(m, R, pk_a)$ 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 $\text{Enc}(m, R', pk_a)$ for some fresh randomness $R' \in \mathsf{R}_{\mathsf{hon}}$. The two values are indistinguishable by (iii) in §2.6.1.

(S3) Encryptions created by dishonest accounts: Such encryptions have the form $\text{Enc}(m, R, pk_a)$ 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 $\bar{t}$ if they are part of $\bar{\sigma}$. Hence, the simulator can simulate these values by copying them from $\bar{\sigma}$. The copied values are indistinguishable by (i) in §2.6.1.

(S4) NIZK proofs: Given $v_{1:n}$, a valid proof $\text{Proof}_\phi(R; v_{1:n}; v'_{1:m})$ can be simulated by constructing $\text{SimPr}_\phi(R'; v_{1:n})$ for fresh $R' \in \mathsf{R}_{\mathsf{hon}}$. These proofs are indistinguishable by (iv) in §2.6.1.

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 $\bar{t}_2$ by respecting the three changes (i–iii) described above and simulating encryptions and NIZK proofs using S1–S4.

Creating a trace indistinguishable from $\bar{t}$ requires consistently reproducing repetitions of equal values (see the bijection π in the definition of trace indistinguishability, §2.6.1). 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 repetitions by remembering previously simulated values for each runtime location.

We now describe how Sim extends $\bar{t}_2$ when following T in C step by step. All entries in $\bar{t}$ 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 $\bar{t}$ 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. 2.9b). 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 $\mathbf{verify}_\phi$ 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 $\mathbf{verify}_\phi$ 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. A.4). □

2.7 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. 2.11, 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 [61]. In the following, we discuss how to instantiate our approach for Solidity.

NIZK PROOFS We use ZoKrates [46] (commit `224a7e6` with proving scheme GM17 [44]) 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 not supported by the used version of ZoKrates. We reflect these restrictions

7 The code is available on GitHub: `https://github.com/eth-sri/zkay/tree/ccs2019`

Contract	Domain	Short description	#fns (priv)	#loc	#loc transf (sol+zok)	#crossings	scen. #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. 2.1)	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 2.1: 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.

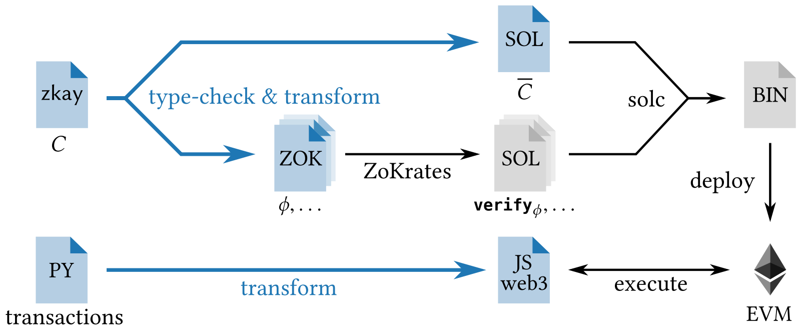


FIGURE 2.11: Overview of implementation.

in zkay's type system by disallowing such operations for private values. We still support them for public values by modifying T_ϕ (Fig. 2.9c) to use *in* for such 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 The used version of ZoKrates does not support asymmetric encryption. Therefore, the proof-of-concept implementation evaluated in this chapter relies on the (insecure) surrogate functions for encryption ($\text{Enc}(v, R, k) = v + k$) and decryption ($\text{Dec}(c, k) = c - k$). Still, later improvements of our implementation (described in the technical report [62]) added support for real encryption, which have then also been integrated in the implementation of zkay's successor ZeeStar (presented in Chapter 3). As we argue in §2.8 and empirically demonstrate in §3.6, changing surrogate encryption to real encryption does not significantly affect the performance of zkay, because the verification gas cost is essentially independent of the encryption function and the off-chain cost for proof generation can only grow moderately. This is because our implementation applies a standard reduction [52] 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.

2.8 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. 2.1. 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 [15], privacy concerns are a key roadblock preventing their adoption [14].

When implementing the contracts in Tab. 2.1, 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. 2.1 shows that the number of lines increases significantly, on average by a factor of more than 6.

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. That 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. 2.8). We note that it is possible 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. 2.1 (column #crossings) indicate that even for simple contracts such as MedStats (see Fig. 2.1a), 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 [63], and execute small transaction scenarios (last column in Tab. 2.1). We use version 0.5.0 of solc, 5.0.14 of Truffle, and 2.5.5 of Ganache. All experiments were run on a machine with 32 GB RAM and 12 cores at 3.70 GHz. We note

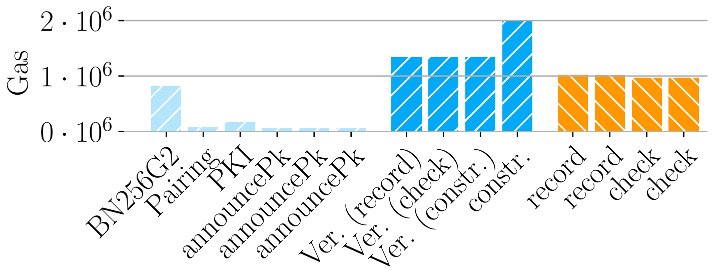


Figure 2.12: On-chain cost for contract `MedStats` (Fig. 2.1).

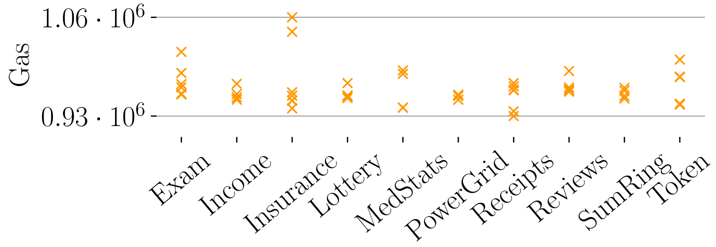


Figure 2.13: On-chain cost for transformed private transactions.

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. 2.1), depicted in Fig. 2.12. Later, we will show how our observations generalize to the other examples. Fig. 2.12 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. 2.12 shows that every transaction costs roughly 10^6 gas. We believe this is a moderate cost for protecting critical data. 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 [46]).

The above observations are general: Fig. 2.13 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 PKCS#1 v2.2 RSA encryption of a message using a 2048-bit key yields a given ciphertext. Generating this proof took roughly 13 seconds, and our generated circuits never used more than 9 encryptions or decryptions.

2.9 DISCUSSION

In this section, we discuss possible extensions of zkay.

OWNERSHIP TRANSFER The zkay language 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 straightforward. 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.

We integrated this extension in later improvements of our implementation, described in the technical report [62].

ALTERNATIVE REALIZATIONS This work leverages encryptions and NIZK proofs to realize zkay contracts. However, we stress that 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 homomorphic encryption (as we will explore in Chapter 3), 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 [44, 45, 53] rely on a trusted setup phase (once per circuit)—a known deployment issue [39]. However, secure multi-party computation (SMC) can be used to reduce trust for this setup phase [64], which is only required once per contract.

COMPATIBILITY WITH EXISTING ANALYSIS TOOLS Various tools enable verification, testing, and other analysis of smart contracts [65, 66, 67]. 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.

2.10 RELATED WORK

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

BLOCKCHAIN PRIVACY As we have already discussed in §1.1, many works bring privacy to payments and transactions using mixers [17, 18, 19, 20, 21] or by cryptographic means [23, 24, 25]. In contrast to all these approaches, zkay brings data privacy to general smart contracts.

Like zkay, Hawk [32], Arbitrum [33], and Ekiden [34] 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 [68]), 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 [39] 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 Private decentralized computation can also be realized using SMC, which hides the inputs of all parties involved in a computation and thereby provides data privacy by construction. However, the execution model of SMC is fundamentally different from that of blockchains: SMC typically requires interactive parties and cannot scale to the number of participants in public blockchains—recent SMC systems handle only up to 150 parties [69, 70]. Further, the common semi-honest attacker model in two-party SMC is weaker than the active attacker model in permissionless blockchains.

Various works explore how privacy-demanding applications can be expressed in high-level languages and compiled to SMC. While zkay targets the instantiation of zk-SNARKs in a blockchain setting, it shares some aspects with languages for SMC.

For instance, similar to zkay, SMCL [71], Wysteria [72], SCVM [73] and ObliVM [74] use types or privacy labels to separate public from secret values. In contrast, zkay additionally indicates the *owner* of private values. While SMCL, SCVM and Wysteria support values stored at only one participant, these values are subject to tampering (i.e., their integrity is not enforced). In contrast, zkay maintains integrity for all values. Similar to zkay, SCVM and ObliVM perform a privacy analysis ensuring information is not implicitly leaked to higher security levels, and provide declassification expressions. While SCVM and ObliVM hide all accessed memory locations, mapping indices are always public in zkay.

Some existing systems including SecreC [75] and ABY [76, 77] use language constructs to indicate the used SMC schemes. In contrast, the privacy types of zkay are not tied to a cryptographic primitive.

Like zkay, many of the above works pose restrictions on control flow and loops in order to prevent leakages based on execution traces. For instance, similarly to zkay, SecreC, SCVM and ObliVM do not support unbounded loops with private conditions. Further, 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.

The recent Viaduct [78] system generalizes the above works by compiling high-level programs into a combination of multiple protocols including SMC and NIZK proofs. Its powerful type system tracks both privacy and integrity constraints, allowing it to combine protocols for semi-honest and active adversaries. However, similarly to other SMC languages and in contrast to zkay, it is not targeted at smart contracts or blockchains.

ZERO-KNOWLEDGE STATEMENTS AS PROGRAMS Multiple systems transform NP statements expressed as high-level programs to zero-knowledge proofs. For instance, Pinocchio [79], Geppetto [80] and Buffet [81] allow private and verifiable execution of programs written in high-level languages. Similarly, TinyRAM [82, 83, 84] and xJsnark [85] provide optimized transformations of high-level proof statements to zk-SNARK circuits.

Compared to zkay, these systems also provide a form of data privacy (by hiding the NP witness), but lack a privacy type system and a blockchain-specific privacy notion. While our implementation of zkay uses ZoKrates [46] to produce zk-SNARKs, its proof circuits could alternatively be expressed in and optimized by xJsnark.

PRIVACY POLICY LANGUAGES JFlow [86] introduces information flow annotations enforcing fine-grained and powerful access control. Jeeves [87, 88] and Jacqueline [89] 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.

2.11 SUMMARY

In this chapter, we presented zkay, a smart contract programming 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 (cp. C1 and C2 in §2.1), 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.

3 AN EXTENSION FOR COMPUTATION ON UNKNOWN DATA

The zkay system presented in the previous chapter does not allow performing computation on unknown data. More technically, an expression such as `x + y` where `x` has type **`uint@me`** and `y` has type **`uint`**`@Bob` (where `Bob` $\neq$ **`me`**) is not supported in zkay. In this chapter, we present an extension of zkay allowing developers to perform this and other operations on unknown data. This extension is key to making our smart contract system applicable for a variety of use cases.

3.1 INTRODUCTION

LIMITATIONS OF ZKAY The zkay system (Chapter 2) ensures data privacy of smart contracts using the following high-level approach: Private values are encrypted for their owner, and updates of encrypted values are enforced to respect the smart contract logic using NIZK proofs.

Unfortunately, a fundamental limitation of this approach is that transactions cannot operate on foreign values (i.e., values owned by parties other than the caller). This precludes zkay from expressing private variants of some of the most popular use cases of Ethereum [90], including private wallets, where coin transfers typically require increasing foreign balances.

THIS CHAPTER: ZEESTAR In this chapter, we address the expressivity restrictions of zkay by complementing it with homomorphic encryption, which allows evaluating specific operations (most importantly, addition) on foreign values.

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

CHALLENGES Integrating homomorphic encryption into zkay is challenging. First, homomorphic encryption and NIZK proofs have incomparable expressivity and must hence be instantiated in combination. For example, realizing a private wallet requires enforcing different ciphertexts to hold the same plaintext encrypted for different parties using a NIZK proof (§3.3.1).

Second, achieving tractable prover efficiency for this combination of primitives is difficult in practice: for instance, combining Groth16 proofs [45] with Paillier encryption [47] leads to an explosion of prover memory and runtime (§3.5.1).

IMPLEMENTATION We implemented ZeeStar as an extension of our zkay implementation presented in §2.7. Our end-to-end tool relies on exponential ElGamal encryption [48] and Groth16 NIZK proofs [45], and uses the idea of elliptic curve embedding from [26, 39] to achieve high prover efficiency. Our evaluation on 12 example contracts demonstrates that ZeeStar is expressive and its costs are comparable to popular existing applications: on average, a ZeeStar transaction costs 339 k gas (see §3.6.4). Further, ZeeStar can readily express the existing confidential payment system Zether [29] at lower gas costs and without requiring familiarity with cryptographic primitives.

OUTLINE The remainder of this chapter is organized as follows.

- After giving an overview of ZeeStar (§3.2), we present how the ZeeStar compiler automatically enforces data privacy while allowing users to perform addition-based modification of foreign values (§3.3).
- In §3.4, we present an extension of ZeeStar to support private multiplication and mixing multiple encryption schemes.
- Next, in §3.5–§3.6 we present our end-to-end implementation[1] of ZeeStar for Ethereum along with an evaluation on 12 contracts.
- Finally, we discuss related work (§3.7) and conclude the chapter (§3.8).

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

3.2 OVERVIEW

In Fig. 3.1, we provide an overview of ZeeStar.

EXAMPLE: PRIVATE TOKENS Fig. 3.1a shows a ZeeStar contract modeling a wallet holding private tokens (cp. Fig. 1.1). Besides the highlighted annotations (discussed shortly) and keyword **`me`** (a shorthand for `msg.sender` in Solidity), the code follows a straightforward Solidity implementation. The mapping `bal` stores the number of tokens held by each individual party. The `transfer` function is used to transfer `val` tokens from the sender **`me`** (Line 5) to another party `to` (Line 6), after checking that the sender has sufficient funds (Line 4). For simplicity, the contract does not contain logic to initialize the balances.

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

PRIVACY ANNOTATIONS AND TYPES To enable precise and ergonomic specification of privacy constraints, ZeeStar relies on the privacy annotations of zkay (Chapter 2). Recall that these annotations are used to track ownership of values in a privacy type system: Data types τ (such as integers and booleans) are extended to types of the form $\tau@\alpha$, where α determines the *owner* of the expression. The value of an expression can only be seen by its owner. The owner α may be **`all`** (indicating the value is public), or an expression of type **`address`**. In this chapter, we call expressions with owner **`me`** to be *self-owned*, and expressions with owner $\alpha \notin \{\texttt{me}, \texttt{all}\}$ to be *foreign*.

In Fig. 3.1a, we highlight the privacy annotations used to model the privacy notion described above. Line 2 specifies that `bal[a]` is private to the address `a`. The argument `val` of type **`uint@me`** (Line 3) is owned by the sender, while `to` of type **`address`** (a shorthand for **`address@all`**) is public.

Like in zkay, in order to prevent implicit information leaks, private expressions with owner α cannot be directly assigned to variables with a different owner $\alpha' \neq \alpha$. Instead, developers can use **`reveal`**`(e, a)` to explicitly reveal a self-owned expression `e` to another owner `a`. For example, in Line 6 we reveal the transferred number of tokens `val` to the recipient `to`. This is needed because `bal[to]` is owned by `to`. To avoid implicit leaks based on access patterns, the control flow of a contract must not depend on any private values. For example, **`require`**`(`e`)` rejects the transaction (i.e.,

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

(a) Input ZeeStar contract with privacy annotations.

```
contract Token {
  mapping(address => bin) bal;
  function transfer(bin val, address to, bin proof,
                    bool b, bin new_me, bin new_to) {
    require(b);
    bal[me] = new_me;
    bal[to] = new_to;
    verifyϕ(proof, ...);
  }
}
```

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

Public inputs:

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

Private inputs (witness):

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

Constraints:

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

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

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

FIGURE 3.1: Compiling an example ZeeStar contract. The proof circuit ϕ relies on homomorphic addition $\oplus$.

aborts and reverts it) if *e* evaluates to false. Thus, Line 4 publicly reveals whether the sender owns at least the number of transferred tokens.

Note that the privacy annotations only induce minimal overhead compared to existing, non-private smart contract languages such as Solidity. As discussed next, privacy is enforced automatically by ZeeStar's compiler, without requiring developers to manually instantiate cryptographic primitives. We note that zkay would reject the contract in Fig. 3.1a, as it cannot increase the foreign value `bal[to]` by `val` (see Line 6).

COMPILATION Like zkay, ZeeStar compiles the input contract to a contract which is executable on Ethereum and enforces the specified privacy constraints. Fig. 3.1b shows a simplified version of the contract generated by ZeeStar for the token contract in Fig. 3.1a.

In the output contract, values with owner $\alpha \neq$ **`all`** are encrypted under the public key of α using an additively homomorphic encryption scheme. Private expressions are pre-computed locally (i.e., off-chain) by the sender, and only published on the blockchain (on-chain) in encrypted form. Expressions revealed to **`all`** are additionally published in plaintext. For example, private expression `bal[`**`me`**`] - val` (Line 5 in Fig. 3.1a) is replaced by a new function argument `new_me` with ciphertext type **`bin`** (Line 6 in Fig. 3.1b), holding the new encrypted balance of the sender. As discussed shortly, ZeeStar uses a NIZK proof to ensure `new_me` is computed correctly. Similarly, Line 6 in Fig. 3.1a is transformed to Line 7 in Fig. 3.1b. Moreover, the revealed result of the comparison in Line 4 (Fig. 3.1a) is replaced by a plaintext argument `b` in Fig. 3.1b.

ENSURING CORRECTNESS To ensure the function arguments `val`, `b`, `new_me`, and `new_to` are computed correctly by the sender, ZeeStar relies on both NIZK proofs and the homomorphic property of the encryption scheme. To this end, for every function, ZeeStar constructs a proof circuit ϕ enforcing correctness. Fig. 3.1c shows the proof circuit for `transfer`. As public inputs, ϕ takes all encrypted function arguments (`val` and new balances), revealed values (`b`), a subset of the previous state of the contract (previous balances), and the public keys of all involved parties. The private inputs consist of secrets known by the sender (most notably, her private key sk_{me}).

Intuitively, any expression involving only public and self-owned variables is computed by the sender as follows: First, decrypt any private input variables. Then, evaluate the expression on the plaintext arguments. Finally, if the expression is private, encrypt the result using the owner's public key. For example, to compute the new balance `new_me`, the sender decrypts her previous balance and the `val` argument, computes the difference, and encrypts the result under her own public key. ZeeStar collects constraints reflecting this computation in the proof circuit ϕ (Fig. 3.1c). Here, $\mathrm{Dec}(x, sk)$ denotes the decryption of x using private key sk.

LEVERAGING HOMOMORPHIC ENCRYPTION Because the encryption scheme is additively homomorphic, ZeeStar also allows evaluating expressions e_1 + e_2 and e_1 - e_2 for e_1, e_2 with owner $\alpha \notin \{\mathbf{me}, \mathbf{all}\}$. For example, the addition of Line 6 in Fig. 3.1a can be evaluated by the sender using the homomorphic operation $\oplus$. First, the sender re-encrypts the plaintext of `val` under the public key of `to` to obtain a ciphertext c. Then, the sender computes `bal[to]` $\oplus\, c$ to obtain `new_to`. In the proof circuit ϕ, ZeeStar ensures that c is computed correctly. Perhaps surprisingly, the operation $\oplus$ is also evaluated inside the proof circuit (see Fig. 3.1c). While this is not required for privacy, it leads to reduced on-chain costs (in fact, as we discuss in §3.5, doing otherwise is infeasible on Ethereum). Further, as we discuss shortly, this allows for greater expressivity.

After constructing ϕ, ZeeStar inserts a proof verification statement into the output contract (see Line 8 in Fig. 3.1b). When calling the `transfer` function, the sender is required to generate and provide a NIZK proof for the circuit ϕ as a function argument `proof`. This is verified by the blockchain in Line 8, where the public arguments of ϕ are provided as arguments to **`verify`** (see "..."). If verification fails, the transaction is rejected and the contract state is reverted.

EXTENSIONS The described design allows for interesting extensions. In §3.4.1, we describe how ZeeStar can also homomorphically evaluate multiplication for most combinations of owners. By repeated application of $\oplus$, the sender can multiply foreign values by a public natural number. Further, because $\oplus$ is evaluated inside the proof circuit, this also applies to self-owned scalars (these simply occur as plaintexts in ϕ). For example, assume `x` is owned by Alice. Bob can multiply `x` by a secret Bob-owned scalar `y`, without revealing `y` to anyone else. This opportunity is unique to the combination of NIZK proofs and additively homomorphic encryption.

In §3.4.2 we will discuss how ZeeStar can be extended to mix homomorphic and non-homomorphic encryption schemes using suitable annotations and a modification of the type system. This is useful as practical homo-

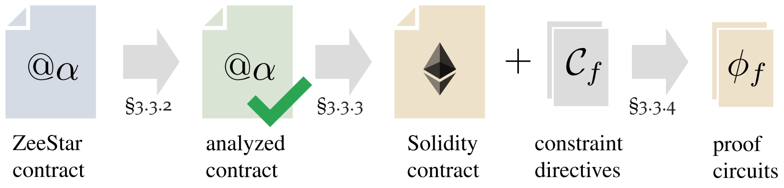


FIGURE 3.2: Compilation steps of ZeeStar.

morphic encryption schemes may come with restrictions (e.g., only 32-bit plaintexts), prompting the developer to only apply these selectively.

3.3 COMPILATION

In this section, we provide a detailed description of ZeeStar. Fig. 3.2 visualizes the three high-level compilation steps. First, the privacy annotations of the input contract are analyzed. Then, the contract is transformed to a Solidity contract and a set of *constraint directives* $\mathcal{C}_f$ for each function f. Finally, each of these sets is transformed to a proof circuit ϕ_f.

Before describing these steps in detail (§3.3.2–§3.3.5), we discuss the key idea of ZeeStar's compilation process.

3.3.1 *Combining NIZK Proofs and Homomorphic Encryption*

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

INCOMPARABILITY OF PRIMITIVES Enforcing correctness in a ZeeStar output contract amounts to ensuring correct computation of ciphertexts (such as `new_me` in Fig. 3.1b). Unfortunately, the two primitives at hand are incomparable in the sense that neither is strictly more expressive than the other. While we can evaluate arbitrary expressions inside proof circuits, using NIZK proofs for correctness generally requires the prover to decrypt all input variables. For instance, the circuit in Fig. 3.1c decrypts the previous balance bal^{me}_{old} of the sender in order to prove correct computation of bal^{me}_{new}. In contrast, additively homomorphic encryption can be used to provide correctness guarantees "by construction," but only for addition and subtraction. For example, the sender does not need to know $sk_{\texttt{to}}$ in order to correctly update `bal[to]` in Fig. 3.1b.

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

FIGURE 3.3: ZeeStar core privacy types α, expressions e and statements S, where c are constants and id are variable identifiers.

In the example of Fig. 3.1, we cannot enforce correctness using only one of the primitives. Relying only on NIZK proofs and non-homomorphic encryption, the sender could not even compute the new balance `new_to`. On the other hand, only using homomorphic encryption is insufficient to guarantee correctness: First, the requirement for sufficient sender funds (Line 4 in Fig. 3.1a) cannot be enforced without some sort of NIZK proof. Second, while the new balance `new_me` of the sender could be updated using $\ominus$, a correct instantiation would still need to somehow enforce that the same value is removed from and added to `bal[me]` and `bal[to]`, respectively (note that the two balances are encrypted under different keys).

KEY IDEA In order to achieve high expressiveness, ZeeStar instantiates the two primitives in combination. ZeeStar's compilation is driven by privacy annotations: for each expression, ZeeStar decides which cryptographic primitive to use, based on privacy types and the actual expression. For example, because `bal[to]` is foreign, ZeeStar determines that adding `val` to it (Line 6 in Fig. 3.1a) requires homomorphic addition. However, because `val` is self-owned, it needs to be re-encrypted under pk_{to} in the proof circuit.

3.3.2 *Privacy Type Analysis*

As a first step, ZeeStar analyzes the privacy annotations in the input (see Fig. 3.2). Before explaining this step in an example, we first discuss the language fragment considered in this chapter in more detail.

LANGUAGE FRAGMENT In this chapter, we focus on the core language fragment shown in Fig. 3.3 (a subset of the zkay language defined in Fig. 2.3). The fragment allows introducing our key ideas without cluttering the presentation.

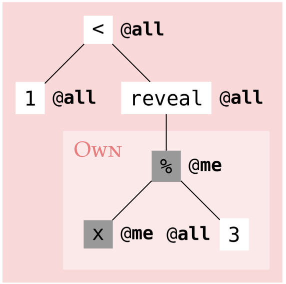
```
 1 contract C {
 2   final address alice; uint@alice a;
 3   final address bob; uint@bob b;
 4   // constructor omitted for simplicity
 5
 6   function f(uint@me x) {
 7     require(alice == me);
 8     require( 1 < reveal(x % 3, all) );
 9     b = (b + reveal(2 * a, bob)) + 4 ;
10     a = x + 1;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

(a) Input contract.

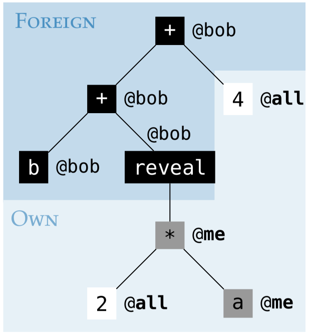
```
function f(bin x, uint e1, bin e2, bin e3, bin p) {
  bin _a = a; bin _b = b;
  require(alice == me);
  require(1 < e1);
  b = e2;
  a = e3;
  verify_φf(p, e1, e2, e3, x, _a, _b, pk(me), pk(bob));
}
```

$$\mathcal{C}_{\mathtt{f}} = \{\mathtt{e1} \equiv_{\mathtt{all}} \mathtt{x \% 3} \tag{3.1}$$
$$\mathtt{e2} \equiv_{\mathtt{bob}} \mathtt{(b + \textbf{reveal}(2 * a, bob)) + 4}, \tag{3.2}$$
$$\mathtt{e3} \equiv_{\mathtt{me}} \mathtt{x + 1}\} \tag{3.3}$$

(c) Transformed function f and collected constraint directives.



(i) Line 8



(ii) Line 9

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

$$\mathtt{e1} = T_{\text{plain}}(\mathtt{x \% 3}) \tag{3.4}$$
$$\mathtt{e2} = T_{\alpha}(\mathtt{(b + \textbf{reveal}(2 * a, bob)) + 4}) \tag{3.5}$$
$$\mathtt{e3} = \text{Enc}(T_{\text{plain}}(\mathtt{x + 1}), pk_{me}, r_0) \tag{3.6}$$

$$\rightsquigarrow$$

$$\mathtt{e1} = \text{Dec}(\mathtt{x}_{old}, sk_{me}) \ \% \ 3 \tag{3.7}$$
$$\mathtt{e2} = (\mathtt{b}_{old} \oplus \text{Enc}(2 \cdot \text{Dec}(\mathtt{a}_{old}, sk_{me}), pk_{bob}, r_1)) \oplus \text{Enc}(4, pk_{bob}, r_2) \tag{3.8}$$
$$\mathtt{e3} = \text{Enc}(\text{Dec}(\mathtt{x}_{old}, sk_{me}) + 1, pk_{me}, r_0) \tag{3.9}$$

(d) Constructing proof circuit constraints.

Figure 3.4: Running example explaining the compilation steps of ZeeStar.

In our fragment, function bodies consist of the statements S shown in Fig. 3.3. Besides sequential composition, these include assignments and `require` statements. The argument to `require` must evaluate to true for the transaction to be accepted. ZeeStar supports standard arithmetic and boolean expressions as well as a dedicated `reveal` expression, which is used to change the owner of a self-owned expression. Variable identifiers (id) include function arguments, contrast fields, local variables, and mapping entries, where the latter is not modeled separately for simplicity. We consider three primitive data types: booleans (`bool`), addresses (`address`), and unsigned integers (`uint`). In ZeeStar, variables can be self-owned (`me`), public (`all`), or owned by a public variable of type `address`.

We can extend ZeeStar to other statements and expressions. In particular, our implementation (see §3.5) accepts a much richer language based on Solidity, including non-recursive function calls (realized by inlining), if-then-else statements (realized by evaluating both branches and multiplexing), and loops. As NIZK proof circuits have bounded size, the latter must be either free from private variables, or manually unrolled up to statically known bounds.

RUNNING EXAMPLE To explain the compilation process of ZeeStar, we use the running example in Fig. 3.4. The code in Fig. 3.4a covers all relevant aspects of compilation but does not implement any meaningful functionality. The fields `alice` and `bob` are initialized in the constructor (not shown) and declared `final` to ensure they are not modified later. Like in zkay, this is used to prevent changing a variable's owner at runtime.

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

STATEMENTS ZeeStar uses the same rules as zkay when analyzing statements. In particular, it requires the argument e of `require(`e`)` to be public, as the fact whether a transaction is accepted leaks the value of e. Also, for assignment statements id $= e$, the owner of both sides must be equal, or e

$$\frac{\Gamma \vdash e_0 \colon \textbf{all} \qquad \Gamma \vdash e_1 \colon \textbf{all}}{\Gamma \vdash e_0 \text{ op } e_1 \colon \textbf{all}}\ \texttt{binop-all}$$

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

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

FIGURE 3.5: Privacy type rules for binary expressions. Here, $\Gamma \vdash e \colon \alpha$ indicates that expression e has privacy type α.

must be public. This allows for implicitly making a public value private, but not implicitly leaking any private values.

EXPRESSIONS Privacy types of expressions are determined recursively. In Fig. 3.4b, we show the privacy types for the subexpressions in Line 8 and Line 9 of Fig. 3.4a. Constants c and the address **me** are public, while the privacy type of variables or mapping entries (id) is determined by their declaration. For example, in Fig. 3.4b-i, `x` has privacy type **me**, while the constant `3` is public. The **reveal**(e, α) expression has privacy type α, where e must be self-owned. Fig. 3.4b-i does not contain a node for **all**, because **all** is formally not an expression (see Fig. 3.3).

For expressions, ZeeStar generally inherits the typing rules of zkay. However, to support operations on foreign data, it uses different typing rules for binary expressions, as we show in Fig. 3.5.

If both operands are public, the result is public (rule `binop-all`). For instance, in Fig. 3.4b-i, the inequality `<` is public as it compares public values.

If one of the operands is self-owned and the other is public or self-owned, the privacy type of the result is set to **me** (rule `binop-me`). This is because the result depends on the private operand, so it should be kept private. For example, the `%` operation in Fig. 3.4b-i has privacy type **me**.

If one operand is foreign, the only applicable operations are addition and subtraction (rule `binop-foreign`), which will later be compiled to $\oplus$ and $\ominus$, respectively. In this case, both operands must have the same owner, or one operand must be public. ZeeStar disallows mixing foreign and self-owned operands to prevent implicit leaks. If mixing is desired, developers can always **reveal** the self-owned operand first.

FINDING SELF-OWNED VARIABLES Sometimes, the owner of a variable is syntactically different from **`me`** but still guaranteed to evaluate to the sender's address at runtime. For example, in Line 9 of Fig. 3.4a, field `a` has privacy type `alice` (by its declaration) but Line 7 ensures that `alice == me`. Like zkay, ZeeStar uses sound static analysis to find such cases. The analysis is based on a few simple, sound, but incomplete rules: For instance, a statement **`require`**`(`*a*`== me)` allows ZeeStar to later substitute *a* by **`me`** as long as *a* is not overwritten. To exploit this, ZeeStar changes the privacy type of fields to **`me`** whenever possible, before determining the privacy type of expressions. For example, `a` has type **`@me`** in Fig. 3.4b-ii.

3.3.3 *Contract Transformation*

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

IDEAL WORLD The input contract specifies executions in an ideal world, where functions are executed according to the semantics of the zkay language (presented in §2.4). In the following, we use $[\![e]\!]$ to denote the plaintext value of an expression e when evaluating it in the ideal world.

CORRECTNESS Intuitively, ZeeStar ensures that in the output contract, the value of any field is encrypted for its owner. More precisely, for any sequence of real-world transactions on the output contract, there exist corresponding ideal-world transactions on the input contract. We formalize this in Thm. 3.1, which assumes that the used NIZK proof system is computationally sound (i.e., it is a zk-SNARG) .

Theorem 3.1 (Correctness). *Assume ZeeStar is instantiated with a zk-SNARG (Def. A.4 in App. A.1). Let $\bar{C}$ be the output contract resulting from the compilation of a well-typed ZeeStar contract C. For any* equivalent *states $\sigma, \bar{\sigma}$ and any transaction $\bar{tx}$, with overwhelming probability: running $\bar{tx}$ on $\bar{C}$ in starting state $\bar{\sigma}$ is either rejected, or there exists a transaction tx for the same function, sender and public arguments as $\bar{tx}$ such that running tx on C in starting state σ results in state σ'* equivalent to *the output state $\bar{\sigma}'$ of $\bar{tx}$.*

Here, we define a state σ to be *equivalent to* a state $\bar{\sigma}$ if both states include the same contract fields, and the (plaintext) value $[\![z]\!]$ of every field z in σ is equivalent to the value of z in $\bar{\sigma}$. We say $[\![z]\!]$ is *equivalent to* a value v iff either

z is public and $v = [\![z]\!]$, or z is owned by $\alpha \neq$ **all** and $v = \text{Enc}([\![z]\!], pk_\alpha, r)$ for some randomness r.

By an inductive argument, for any polynomial-length sequence of transactions on $\bar{C}$ in the empty starting state, with overwhelming probability there exists a corresponding sequence of transactions on C. We will prove Thm. 3.1 in §3.3.5.

PROCESSING A CONTRACT Alg. 3.1 describes how ZeeStar transforms a contract. This algorithm replaces publicly revealed and private expressions by new function arguments, and enforces the latter to respect equivalence as defined above.

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

For example, Fig. 3.4c shows the modified function `f` and the produced $\mathcal{C}_{\mathtt{f}}$ when running Alg. 3.1 on Fig. 3.4a. Copies `_a`, `_b` (Line 2) and the **`verify`** statement (Line 7) are discussed later.

For each expression e occurring in a statement **`require`**(e) or id $= e$ inside the function body, Lines 3–4 (Alg. 3.1) run TRANSFORMEXPR$(e, f, \mathcal{C}_f)$. If e is private to α, it is replaced by a new function argument `arg` (Line 10). Further, ZeeStar adds "`arg` $\equiv_\alpha e$" to $\mathcal{C}_f$, indicating that `arg` should contain the encryption of $[\![e]\!]$ for α. In our example, the whole expression tree in Fig. 3.4b-ii is replaced by a new function argument `e2` with ciphertext type **`bin`** (Line 5 in Fig. 3.4c). We add Eq. (3.2) shown in Fig. 3.4c to $\mathcal{C}_{\mathtt{f}}$. Line 10 in Fig. 3.4a is processed analogously, yielding Line 6 and Eq. (3.3) in Fig. 3.4c.

PUBLIC EXPRESSIONS Note that public expressions e may contain subexpressions of the form **`reveal`**$(e', $ **`all`**$)$, where e' is self-owned. For example, in Fig. 3.4a, the result of the `%` operation is revealed publicly. Hence, Alg. 3.1 performs a top-down tree search (for example, BFS) over public expressions e to find subtrees rooted at **`reveal`**$(e', $ **`all`**$)$ expressions. These are replaced by a new function argument, and an according constraint is added to $\mathcal{C}_f$. In our example, Line 8 in Fig. 3.4a is replaced by Line 4 in Fig. 3.4c and we record Eq. (3.1) in Fig. 3.4c.

Algorithm 3.1 Transforming Function Bodies

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

3.3.4 *Proof Circuit Construction*

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

PROOF CIRCUIT INPUTS First, ZeeStar assembles the public inputs for ϕ_f. These connect the actual values occurring in a transaction with the values in the circuit. For each "$x \equiv_\alpha e$" in $\mathcal{C}_f$, it adds public inputs x and pk_α (if $\alpha \neq$ **all**) to ϕ_f. Further, ZeeStar collects all variables id occurring in e (i.e., function arguments, contract fields, and local variables) and adds, for each id, a public input id$_{old}$ to ϕ_f. [2] Similarly, ZeeStar adds a public input *me* to ϕ_f if **me** occurs in e. To simplify our explanation, we assume that function bodies are in static single assignment form: function arguments cannot be assigned to, contract fields are never read after assignment, and local variables are assigned to exactly once. By the introduction of fresh local variables, any function can be converted to this form. This ensures that constraint directives can be processed independently, and all accesses of a

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

variable id have the same value at runtime, accessible via id_{old} in the proof circuit. In our running example, by Eqs. (3.1)–(3.3), the public proof inputs are: `e1`, `e2`, `e3`, x_{old}, a_{old}, b_{old}, pk_{me}, pk_{bob}.

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

To pass the actual values of the public circuit inputs to ϕ_f, ZeeStar adds a proof verification statement to the output contract. To this end, the previous values of any overwritten fields are copied at the beginning of the function. For example, in Fig. 3.4c, ZeeStar first copies the old values of `a` and `b` in Line 2. In Line 7, it introduces a verification statement accepting the proof `p` and all public proof inputs. Here, **`pk`**`(`α`)` fetches the public key of α from a public key infrastructure.

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

More precisely, we can partition the nodes of e into two sets FOREIGN and OWN, where (i) FOREIGN contains all nodes with owner $\alpha \notin \{\texttt{me}, \texttt{all}\}$, (ii) OWN contains all nodes with owner $\alpha \in \{\texttt{me}, \texttt{all}\}$, and (iii) the subgraph induced by FOREIGN is connected and, if non-empty, contains the root. Conceptually, this divides the expression tree into an upper part FOREIGN and a lower part OWN. For example, in Fig. 3.4b-ii, OWN contains the nodes $*$, `2`, `a`, and `4`. If the root is self-owned, then FOREIGN $= \emptyset$, as for the expression tree of constraint directive `e1` $\equiv_{\texttt{all}}$ `x % 3` (rooted at `%` in Fig. 3.4b-i).

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

TRANSFORMING EXPRESSIONS We now define two recursive transformation functions T_{plain} and T_α used to build constraints for ϕ_f from expressions.

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

FIGURE 3.6: Transforming constraint directives to constraints.

$$T_{\text{plain}}(c) = c \tag{3.10}$$
$$T_{\text{plain}}(\texttt{me}) = me \tag{3.11}$$
$$T_{\text{plain}}(e_1 \text{ op } e_2) = T_{\text{plain}}(e_1) \text{ op } T_{\text{plain}}(e_2) \tag{3.12}$$
$$T_{\text{plain}}(\texttt{reveal}(e, \alpha)) = T_{\text{plain}}(e) \tag{3.13}$$
$$T_{\text{plain}}(\text{id}) = \begin{cases} \text{id}_{old} & \text{if id public} \\ \text{Dec}(\text{id}_{old}, sk_{me}) & \text{otherwise} \end{cases} \tag{3.14}$$

FIGURE 3.7: Recursive expression transformation using T_{plain}.

The function T_{plain} is used to process nodes in OWN. It is designed such that for any $e \in$ OWN, evaluating $T_{\text{plain}}(e)$ inside the proof circuit results in $[\![e]\!]$. On the other hand, T_α targets nodes in FOREIGN and nodes in OWN whose parents are in FOREIGN. For expression e, evaluating $T_\alpha(e)$ inside the proof circuit results in the ciphertext $\text{Enc}([\![e]\!], pk_\alpha, r_i)$ for some randomness r_i.

Before discussing T_{plain} and T_α in detail, we describe how they are used. Specifically, ZeeStar transforms each constraint directive "$x \equiv_\alpha e$" in $\mathcal{C}_f$ to a constraint in ϕ_f enforcing equivalence. Depending on α, the constraint has a different form as shown in Fig. 3.6. If $\alpha = \texttt{all}$, T_{plain} enforces that x holds the plaintext value of e. This ensures that self-owned values are correctly revealed by `reveal(`e`, all)`. If $\alpha = \texttt{me}$, x should contain the encryption of $[\![e]\!]$ (determined using T_{plain}) under the sender's public key pk_{me} and some randomness r_i which is added to the private inputs. The third case deals with expressions for which FOREIGN is non-empty. In this case, x is owned by a party $\alpha \neq \texttt{me}$ and we leverage T_α to ensure x contains the correctly encrypted value. In our running example, based on Eqs. (3.1)–(3.3), ZeeStar adds constraints (3.4)–(3.6) in Fig. 3.4d to ϕ_f, where r_0 is a new private input.

$$T_\alpha(c) = Enc_\alpha(c) \tag{3.15}$$

$$T_\alpha(\texttt{me}) = Enc_\alpha(me) \tag{3.16}$$

$$T_\alpha(\text{id}) = \begin{cases} \text{id}_{old} & \text{if id owned by } \alpha \\ Enc_\alpha(\text{id}_{old}) & \text{else if id public} \\ \perp & \text{otherwise} \end{cases} \tag{3.17}$$

$$T_\alpha(\underbrace{e_1 \text{ op } e_2}_{=:e}) = \begin{cases} Enc_\alpha(T_{\text{plain}}(e)) & \text{if } e \text{ public} \\ T_\alpha(e_1) \oplus T_\alpha(e_2) & \text{else if op} = \texttt{+} \\ T_\alpha(e_1) \ominus T_\alpha(e_2) & \text{else if op} = \texttt{-} \\ \perp & \text{otherwise} \end{cases} \tag{3.18}$$

$$T_\alpha(\texttt{reveal}(e, \alpha')) = \begin{cases} Enc_\alpha(T_{\text{plain}}(e)) & \text{if } \alpha = \alpha' \\ \perp & \text{otherwise} \end{cases} \tag{3.19}$$

$$\text{where} \quad Enc_\alpha(e) := \text{Enc}(e, pk_\alpha, r_i) \tag{3.20}$$

FIGURE 3.8: Recursive expression transformation using T_α. Undefined cases ($\perp$) never apply for well-typed contracts.

PLAINTEXT EVALUATION Fig. 3.7 defines the function T_{plain}. At a high level, this function decrypts any self-owned variables occurring in e and recursively evaluates the expression (i.e., it works analogously as zkay). The rules for constants, **me**, and binary operations are straightforward. By Eq. (3.13), reveal expressions are ignored (these are only used for preventing implicit leaks). Eq. (3.14) shows the rule for transforming a variable id. If the variable id is public, id_{old} can be accessed directly. Otherwise, as the contract is well-typed and T_{plain} is applied only to nodes in OWN, the value is self-owned and is hence decrypted using sk_{me}. For example, in Fig. 3.4d, Eq. (3.4) is transformed to Eq. (3.7).

HOMOMORPHIC EVALUATION Fig. 3.8 defines T_α, which produces values encrypted for α. Constants and **me** are public, hence their plaintext value is encrypted under the public key of α using the function Enc_α (Eqs. (3.15)–(3.16) and Eq. (3.20)). Here, r_i is a new private input for ϕ_f. For foreign variables id, ZeeStar accesses id_{old}, which holds a ciphertext for α (as the contract is well-typed, T_α is never applied to private variables with owner $\neq \alpha$). If id is public, then it is encrypted for α.

For binary operations (Eq. (3.18)), we distinguish multiple cases. If the operation is public, then we compute its plaintext value using T_{plain} and again apply Enc_α. Private additions and subtractions are computed homomorphically: before applying $\oplus$ or $\ominus$, the arguments are recursively transformed by T_α to obtain two ciphertexts encrypted for α. Well-typed contracts do not involve other binary operations on foreign arguments.

For well-typed contracts, T_α is only applied to nodes in FOREIGN and their direct children. Hence, an expression **`reveal`**(e, α') is only reachable by T_α if $\alpha' = \alpha$, and we can apply T_{plain} and Enc_α (see Eq. (3.19)). Conceptually, this and all other cases introducing Enc_α provide a "bridge" between FOREIGN and OWN. Note that public expressions can be mixed with foreign expressions using `+` or `-` : for example, the constant `4` in Fig. 3.4b-ii is in OWN but is an argument to the root `+` in FOREIGN. Hence, Eq. (3.15) introduces a bridge for `4`.

In the example of Fig. 3.4d, Eq. (3.5) is transformed to Eq. (3.8), where r_1, r_2 are new private inputs of ϕ_f.

Algorithm 3.2 Transforming Transactions

```
1: procedure $T_{\text{TX}}(C, \bar{\sigma}, tx, sk_{me}, \Sigma, \text{pk})$
2:     Initialize transaction $\bar{tx}$ for same function and sender as $tx$
3:     Copy values of public arguments from $tx$ to $\bar{tx}$.
4:     for each private argument with value $v$ in $tx$ do
5:         Add freshly encrypted argument $\text{Enc}(v, \text{pk}(me), r)$ to $\bar{tx}$
6:     for each argument arg introduced by Alg. 3.1 in $C$ do
7:         Compute the value $x$ of arg according to the rules in
           Fig. 3.6, using $sk_{me}$ to decrypt self-owned fields in $\bar{\sigma}$
           and public keys in pk to encrypt values for other
           parties
8:         Add $x$ to $\bar{tx}$
9:     Use $\Sigma$ to generate the NIZK proof $\pi$ and add $\pi$ to $\bar{tx}$
10:    return $\bar{tx}$
```

TRANSACTION TRANSFORMATION To call a function f in the transformed contract, the sender needs to prepare the arguments introduced by ZeeStar. More precisely, let C be an input ZeeStar contract, and $\bar{C}$ the transformed output contract. In order to call a function f on $\bar{C}$ in a starting state $\bar{\sigma}$, a sender first assembles a transaction tx for the input contract C, where tx indicates f, the sender address $\text{sender}[tx]$, and the arguments to f. Let Σ be the cryptographic parameters of the NIZK proof system. In the case of a zk-SNARG (Def. A.4 in App. A.1), this includes the common reference string (CRS) generated by the Setup algorithm. Further, let pk be a table mapping accounts to their public keys. The sender runs $T_{\text{TX}}(C, \bar{\sigma}, tx, sk_{\text{sender}[tx]}, \Sigma, \text{pk})$ as shown in Alg. 3.2 to create a transaction $\bar{tx}$ for $\bar{C}$. At a high level, this function selects the public arguments of ϕ_f such that ϕ_f is satisfied and generates a NIZK proof for ϕ_f.

Our implementation (§3.5) includes a transparent interface performing these steps automatically.

3.3.5 *Discussion*

CORRECTNESS ZeeStar satisfies correctness, as stated in Thm. 3.1.

Proof sketch (Thm. 3.1). First, we prove that sk_{me} and pk_{me} inside the proof circuit belong to the transaction sender with overwhelming probability. As the blockchain authenticates the sender of a transaction (e.g., via a signature in Ethereum), the contract $\bar{C}$ ensures that pk_{me}, which is passed as a public input for proof verification, belongs to the original sender. If the transaction is accepted, the zk-SNARG is successfully verified by an honest verifier. Therefore, by the computational soundness property, a private input sk_{me} such that ϕ is satisfied must exist with overwhelming probability. As sk_{me} is enforced to correspond to pk_{me} in ϕ (e.g., see Fig. 3.1c), it is guaranteed to belong to the original sender with overwhelming probability.

Given that sk_{me} and pk_{me} are correct, the correctness of state updates follows from the computational soundness of the zk-SNARG and inductive reasoning on the transformation rules (Figs. 3.6–3.9), which ensure correctness by construction via the constraint directives $x \equiv_\alpha e$. □

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

PRIVACY ZeeStar satisfies the following notion of privacy:

Theorem 3.2 (Privacy, informal). *Let $\bar{C}$ be the output contract resulting from the compilation of a well-typed ZeeStar contract C. An active attacker cannot* *learn more from real transactions on $\bar{C}$ than from the information observable in the corresponding ideal-world transactions on C (see §3.3.3).*

We prove a more formal version of Thm. 3.2 in App. A.3, assuming that ZeeStar is instantiated with an IND-CPA encryption scheme and a computationally sound and perfectly zero-knowledge NIZK proof system (a weaker notion of zk-SNARK formalized as zk-SNARG in App. A.1). In a standard simulation-based proof, we use a hybrid argument to show that for any probabilistic polynomial-time (PPT) adversary statically corrupting a set of parties, any sequence of real-world transactions is computationally indistinguishable from transactions simulated from information available to the adversary in the ideal world. As ZeeStar is an extension of zkay, this proof supersedes the symbolic proof presented in §2.6.

LIMITATIONS ZeeStar is limited by the expressiveness of proof circuits and additively homomorphic encryption. Specifically, as proof circuits are bounded, ZeeStar contracts cannot access unbounded amounts of private memory or include unbounded loops with private operations. However, this is not a concern in practice: Due to the *block gas limit* in Ethereum, which bounds the computation of a transaction, using unbounded loops is discouraged [91]. Instead, elements of an unbounded data structure should be processed in individual transactions.

Further, foreign values can only be subject to addition or subtraction where either both operands are owned by the same party, or one operand is self-owned or public. In §3.4.1, we discuss how to alleviate this restriction by allowing also multiplication for most combinations of owners.

3.4 EXTENSIONS

Next, we show how ZeeStar can homomorphically evaluate multiplications for most combinations of owners (§3.4.1), and how different encryption schemes can be mixed (§3.4.2).

3.4.1 *Homomorphic Multiplication by Known Scalars*

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

$$T_\alpha(e_0 * e_1) = \oplus^{T_{\text{plain}}(e_1)}\, T_\alpha(e_0) \tag{3.21}$$

$$T_\alpha(e_0 * \texttt{reveal}(e_1, \alpha)) = (\oplus^{T_{\text{plain}}(e_1)}\, T_\alpha(e_0)) \oplus \text{Enc}(0, pk_\alpha, r_i) \tag{3.22}$$

FIGURE 3.9: Expression transformation rules for homomorphic scalar multiplication, where e_0 is foreign and e_1 is public (Eq. (3.21)) or self-owned (Eq. (3.22)). Symmetric rules omitted.

MULTIPLICATION BY PUBLIC SCALARS The compilation process described in §3.3 can easily be extended to support homomorphic multiplication of foreign values by public scalars. In particular, the privacy type rule `binop-foreign` (Fig. 3.5) is extended to allow $e_0 * e_1$ for foreign e_0 with owner α and public e_1 (or vice-versa) and assign privacy type α to the result. When transforming expressions, ZeeStar performs this multiplication homomorphically inside the proof circuit. To this end, we extend T_α by rule (3.21) in Fig. 3.9.

For example, the contract in Fig. 3.4a would still compile if we replaced `+ 4` in Line 9 by `* 4`. In this case, by Eq. (3.21), Eq. (3.8) in Fig. 3.4d would change to

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

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

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

Unfortunately, naively applying Eq. (3.21) to this case leads to a privacy leak. Consider Alice producing $y = \oplus^s\ x$, where x is encrypted for Bob and

s is a private scalar owned by Alice. If an adversary Eve knows y and x, she can enumerate the potentially small space of possible scalars s' and find s by checking if $y = \oplus^{s'} x$. To prevent this attack, Alice must re-randomize y using fresh randomness before publishing y. To this end, when constructing the proof circuit, ZeeStar re-randomizes the product $z := \oplus^{T_{\text{plain}}(e_1)} T_\alpha(e_0)$ by homomorphically adding a freshly encrypted constant 0 to z. This is formalized in Eq. (3.22) of Fig. 3.9. Here, we assume the additional property that $\text{Enc}(x, pk, r) \oplus \text{Enc}(0, pk, r')$ is indistinguishable from a fresh encryption $\text{Enc}(x, pk, r'')$ for any x, pk, r. This property is formalized in App. A.1. As we show in App. A.3, the above transformation preserves privacy.

DISCUSSION At a high level, this extension allows homomorphically multiplying two ciphertexts using an additively homomorphic encryption scheme, as long as one of these is encrypted for the sender. This is unique to the combination of NIZK proofs and additively homomorphic encryption: without the former, we could not guarantee correctness of the result. In §3.6, we show how such multiplications can be used to implement 1-out-of-2 oblivious transfer.

3.4.2 *Mixing Homomorphic and Non-homomorphic Schemes*

In practice, homomorphic encryption schemes are often subject to restrictions. For example, exponential ElGamal encryption [48] only supports short plaintexts (≈ 32 bits; see §3.5.1). Therefore, it can be useful to use non-homomorphic encryption where possible and only selectively apply homomorphic encryption where needed. We now discuss an extension of ZeeStar which allows mixing such schemes.

HOMOMORPHISM TAGS We extend ZeeStar's privacy annotations by *homomorphism tags* of the form <μ> for $\mu \in \{+, \text{␣}\}$, where μ determines the homomorphic property of the encryption scheme. In particular, when declaring a variable, the developer adds a tag of the form <μ>, specifying whether the variable should be encrypted using an additively homomorphic scheme (by <+>) or a non-homomorphic scheme (by < >, or no tag). For example, in the contract of Fig. 3.4a, the field a can be encrypted non-homomorphically as it is never subject to foreign addition, by specifying the following tags:

```
2 final address alice; uint@alice a;
```

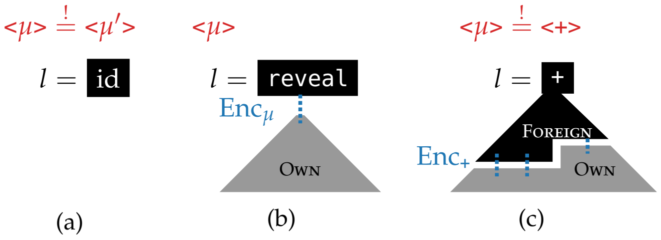


Figure 3.10: Cases for determining encryption scheme.

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

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

To this end, we adapt (i) ZeeStar's proof circuit construction (§3.3.4) to automatically select the appropriate encryption and decryption functions when processing a constraint directive, and (ii) ZeeStar's privacy analysis (§3.3.2) to only accept contracts admitting a non-conflicting selection. Decryption is only introduced by Eq. (3.14), where the function Dec_μ is determined by the homomorphism tag `<`μ`>` of the variable.

SELECTING THE ENCRYPTION FUNCTION Selecting the encryption function Enc_μ is more interesting. If $\alpha =$ **`all`** for a directive $x \equiv_\alpha e$, no encryption is needed. Otherwise, the directive must originate from Line 11 in Alg. 3.1 and e must therefore be the right-hand side of an assignment $l = e$. Below, we distinguish the possible cases for α.

If $\alpha =$ **`me`**, the second case in Fig. 3.6 applies. The used encryption function Enc_μ is then determined by the tag `<`μ`>` of l. For example, for Line 10 in Fig. 3.4a, ZeeStar uses Enc to encrypt the result of `x + 1` because `a` is declared as `@alice`. Note that this allows for implicitly switching encryption schemes of self-owned values: an assignment $l = e$ is accepted by ZeeStar even if l is annotated `@me<+>` and e contains variables annotated as `@me` (or vice-versa). For instance, if `x` in Fig. 3.4a was declared as `@me<+>`, the code would still compile.

Otherwise ($\alpha \notin \{$**`all`**, **`me`**$\}$), we distinguish the three cases visualized in Fig. 3.10. If e is a foreign variable id (Fig. 3.10a), no encryption operation is introduced as id is already encrypted. To enable this, we adapt the privacy analysis (§3.3.2) to raise a type error if the tag $<\mu'>$ of id does not match the tag $<\mu>$ of l. If e is a `reveal` expression (Fig. 3.10b), e is processed by Eq. (3.19). Then, the encryption scheme used in Enc_α is selected to match the homomorphism tag $<\mu>$ of l. Otherwise, e must be an addition or subtraction expression (Fig. 3.10c), to be evaluated in ϕ_f using $\oplus$ or $\ominus$ by Eq. (3.18). Using $\oplus$ or $\ominus$ requires their arguments to be ciphertexts under Enc_+, which recursively applies to all + and - nodes in Foreign. Therefore, we adapt the privacy analysis to reject private variables in Foreign which do not have tag <+>, and instantiate Enc_α using Enc_+ for all bridges to Own (see Fig. 3.8). Further, ZeeStar ensures that the left-hand side l has tag <+>.

3.5 IMPLEMENTATION

We now present our implementation of ZeeStar.

3.5.1 *Efficient Cryptographic Operations*

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

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

PROBLEM: HIGH COSTS Reducing ϕ to operations over $\mathbb{F}_q$ can lead to high emulation overhead for some operations (e.g., for computation over a field $\mathbb{F}_p \neq \mathbb{F}_q$), resulting in prohibitively high proof generation costs. For instance, generating a Groth16 [45] zk-SNARK for Paillier encryption [47] with 2048-bit keys requires over 256 GB of RAM—an impractical requirement for commodity desktop machines.

SOLUTION: CURVE EMBEDDING To address this issue, we instead leverage an encryption scheme based on an elliptic curve E_2 (discussed shortly). This allows us to rely on *curve embedding* [26, 39], which reduces prover costs for elliptic curve operations inside proof circuits. In this technique,

E_1 and E_2 are carefully selected such that the base field of E_2 equals $\mathbb{F}_q$, where q is determined by E_1. This allows operations on E_2 to be evaluated natively in $\mathbb{F}_q$, without emulation overhead.

Because at the time of this writing, Ethereum only provides precompiled contracts for the BN254 curve [55, 92] and proof verification involves operations over E_1, we use Groth16 [45] zk-SNARKs over $E_1 = \text{BN254}$. For E_2, we use the Baby Jubjub curve [93], whose base field matches the scalar field of BN254. This choice allows for efficient cryptographic operations in ϕ.

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

SETUP FOR ZK-SNARKS Like other systems relying on Groth16 [39], our implementation is subject to a circuit-specific trusted setup phase. This setup can for instance be executed using SMC [64].

Still, we stress that ZeeStar is fundamentally not limited to zk-SNARKs with a trusted setup. For instance, we could instantiate Bulletproofs [94] to trade the trusted setup for increased verifier complexity. Recently, several more efficient proving schemes with universal [95, 96, 97, 98] or transparent setup [99] have been proposed. Once practical for Ethereum, these can likely replace the Groth16 zk-SNARKs in ZeeStar.

HOMOMORPHIC ENCRYPTION To leverage the benefits of curve embedding, our implementation relies on exponential ElGamal encryption [48] over the Baby Jubjub curve [93]. As discussed in App. A.4, this scheme is additively homomorphic, provides a closed-form formula for scalar multiplication, and supports re-randomization (as required by the extension from §3.4.1).

In this scheme, decryption requires solving a discrete logarithm (see App. A.4). For small plaintext lengths k, this can be computed efficiently[3] using the baby-step giant-step algorithm [100]. However, decryption is generally intractable if k is large. Therefore, like previous works [29, 31], we restrict the plaintext to $k = 32$ bits for this encryption scheme. Longer plaintexts can still be encrypted non-homomorphically using the extension presented in §3.4.2. In our evaluation (§3.6), a single decryption never takes longer than 7 s.

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

Even when restricting the plaintext size at encryption sites, the plaintext underlying a homomorphic addition $x \oplus y$ may still exceed 32 bits. Note that not knowing both x and y, there is generally no way for the sender to detect this. Like other work [29], we assume that application-specific logic is used to prevent such overflows (and similarly, underflows at 0). For example, when initializing the balances in Fig. 3.1a, we can use `require` statements to enforce the sum of *all* balances in `bal` to be less than 2^{32}. To make developers aware of this caveat, the type system of our implementation distinguishes integers of different bit sizes (e.g., `uint32` and `uint64`), and restricts the homomorphic tag <+> to be only used with $\leq$ 32-bit integers.

3.5.2 *ZeeStar for Ethereum*

We implemented ZeeStar for Ethereum, including the extensions from §3.4, by extending our implementation of zkay presented in §2.7.

Our end-to-end system accepts Solidity code with privacy annotations and produces (i) a contract executable on Ethereum, and (ii) a transaction interface allowing to transparently interact with ZeeStar contracts. It relies on jsnark [101] and libsnark [102] to generate zk-SNARKs. For each proof circuit, we generate a separate proof verification contract. We use solc v0.6.12 to compile Solidity code and web3 v5.19 to interact with Ethereum.

We use ElGamal encryption as described in §3.5.1 with 251 bit keys. For non-homomorphic encryption, we use the hybrid ECDH Chaskey cipher (previously introduced to zkay as part of the extension described in [62]) with 253 bit keys.

3.6 EVALUATION

Next, we evaluate our implementation presented in §3.5. All our experiments are conducted on a machine with 32 GB of RAM and 12 CPU cores at 3.70 GHz. We use the eth-tester v0.5.0b4 backend ("Berlin" upgrade) to simulate transactions.

3.6.1 *Example Contracts*

We used ZeeStar to implement 12 contracts shown in Tab. 3.1. Contracts reviews and token are homomorphic variants of the examples with the same names in Tab. 2.1. Zether-confidential and zether-large are based on Zether [29] (discussed shortly). The other contracts were introduced by

TABLE 3.1: Contracts used in the evaluation. We specify the number of code lines (LoC), and whether the contracts use homomorphic addition ($\oplus$), homomorphic scalar multiplication ($\oplus^s$, see §3.4.1), or mix encryption schemes ($<\mu>$, see §3.4.2).

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

us. All contracts involve operations on foreign data and hence cannot be expressed by zkay. Below, we discuss two contracts in more detail.

ZETHER CONFIDENTIAL TRANSACTIONS Zether [29] proposes a confidential transaction contract for Ethereum, based on additively homomorphic encryption and NIZK proofs. The contract holds encrypted balances in a table and allows sending a secret amount to another party. To prevent front-running attacks, it maintains a separate "pending" state which is used to receive currency and is periodically rolled over into the balance table. Zether allows "locking" an account to a contract such that only this contract can spend the account's balance.

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

OBLIVIOUS TRANSFER 1-out-of-2 oblivious transfer [103] is a protocol for sending one out of two messages x_0, x_1 to a receiver. The receiver can choose i to learn x_i, without learning the other message x_{1-i} and without revealing i to the sender.

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

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

Here, recv is the address of the receiver, and m is a field owned by recv. The function uses homomorphic scalar multiplication by self-owned values $\texttt{x}_0$, $\texttt{x}_1$. The messages $\texttt{x}_i$ are both marked as being revealed to the receiver, but the receiver only learns the result of the sum: due to Eq. (3.18), the sum is computed inside the proof circuit. Because exactly one of the bits $\texttt{b}_i$ is 1, this is either $\texttt{x}_0$ or $\texttt{x}_1$. Note that revealing both $\texttt{x}_0$ and $\texttt{x}_1$ is unavoidable in our type system because the receiver could potentially learn both $\texttt{x}_0$ and $\texttt{x}_1$ (but not at the same time).

3.6.2 *Compilation and Setup Performance*

We analyze the performance of ZeeStar by compiling each example in Tab. 3.1. In addition to the steps described in §3.3, this includes, for each proof circuit, a zk-SNARK setup phase (see §3.5.1). Compilation takes 66.9 s per contract on average and requires at most 3.07 GB RAM. Runtime is dominated by the setup phases (91% of the time on average). As the setup time depends on the number and the sizes of proof circuits, compilation time varies between 26.4 s (shared-prod) and 144.1 s (zether-large). These are one-time costs per contract.

3.6.3 *Transaction Generation Performance*

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

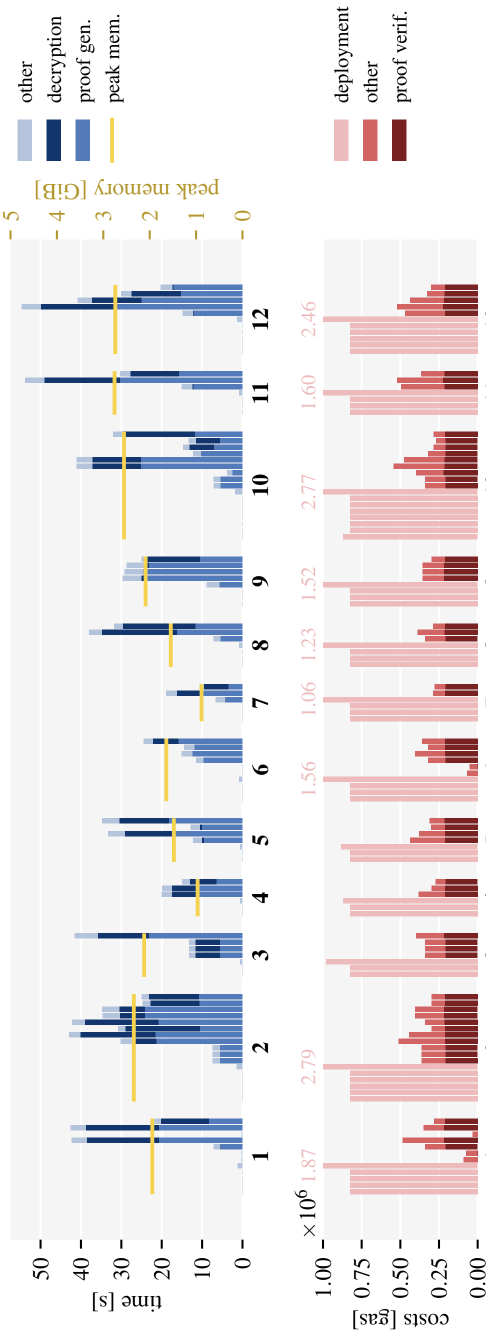


FIGURE 3.11: Evaluation results for example scenarios. Each numbered group corresponds to a series of transactions on an example contract (see No. in Tab. 3.1). Top: Runtime and memory for transaction generation. Bottom: Gas costs for transaction execution.

TABLE 3.2: Number of rank-1 constraint system (R1CS) constraints for cryptographic operations.

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

SCENARIOS For each contract in Tab. 3.1, we prepare a short sequence of transactions called *scenario*, which includes transactions for deploying the verification contracts (§3.5.2) and executing the main contract constructor, but omits the one-time effort of deploying the public key infrastructure. For example, scenario 5 consists of three deployment transactions and two rounds of oblivious transfer (two transactions each).

RESULTS In Fig. 3.11 (top), we show the runtime and peak memory of transaction generation for all scenarios. Each bar shows the runtime of one transaction, separately indicating the runtimes of proof generation and decryption of ciphertexts (which includes solving a discrete logarithm, see §3.5.1).

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

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

PROOF CIRCUIT SIZE In order to better understand the proof generation time, Tab. 3.2 indicates which operations in the proof circuit are most expensive. Specifically, for each cryptographic operation of the ElGamal encryption scheme presented in §3.5.1, Tab. 3.2 shows the number of generated rank-1 constraint system (R1CS) constraints. As proof generation time is linear in the number of R1CS constraints, this number is a good indicator for the cost of each operation.

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

3.6.4 *Transaction Execution Gas Costs*

Transactions on Ethereum are subject to gas costs. We next measure these costs for the transactions generated in §3.6.3.

RESULTS Fig. 3.11 (bottom) shows the gas costs for each transaction, again grouped by scenario. Deployment transactions include deployments of the verification contracts and the main contract constructors. The gas costs of such transactions are relatively high because the sender has to pay for storing the contracts' byte code. However, these are one-time costs per contract instance. For each scenario, the highest cost is induced by the main contract constructor. The overall highest costs of 2.79 M and 2.77 M are observed for inheritance and weighted-lottery, resp., which are the two largest contracts (see Tab. 3.1).

For all non-deployment transactions, we separately indicate the costs induced by proof verification. Because the complexity of zk-SNARKs verification is essentially constant (see §1.3.1), these gas costs are very similar across all transactions involving a NIZK proof. The remaining costs vary between transactions, which we believe is due to the varying number of costly storage operations. On average, a non-deployment transaction costs 339 k gas. The highest non-deployment gas cost of 544.44 k is observed for weighted-lottery.

These costs are comparable to existing applications: A transaction on Uniswap [11] (on 2021-07-14, this was the top application on the ETH25 leaderbord[4]) frequently costs more than 250 k gas. [5]

4 According to `https://www.ethgasstation.info/` (accessed: 2021-07-21).

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

COMPARISON TO ZETHER In [29], a Zether confidential transfer is reported to use 7 188 k gas. In contrast, the analogous ZeeStar transaction on zether-confidential only requires 521.22 k gas.

The Ethereum gas cost model has changed since the publication of [29], so we cannot directly compare these numbers. We have repeatedly tried to contact the authors to provide us the contract code or assist us with updated numbers, but we unfortunately did not receive a response. Zether relies on the Σ-*Bullets* proof system, which does not require a trusted setup but has high verifier costs. For cases where a trusted setup phase is acceptable (potentially based on SMC, see §3.5.1), ZeeStar can offer significantly lower gas costs than Zether.

MONETARY COSTS Transaction fees are computed by multiplying gas costs by the *gas price*, determined by supply and demand. At the time of writing, the gas price is extremely volatile: Even within a *single day*, the recommended gas price[6] fluctuates between 9 and 901 Gwei (2021-07-23). Thus, depending on the time, an average ZeeStar transaction (339 k gas) would have cost between 6.18 and 618.51 USD on this day (for 1 ETH = 2 025 USD). Hence, it is currently impossible to provide useful cost estimates for ZeeStar transactions.

The high and volatile transaction fees of Ethereum are a well-known problem [104, 105], which should be solved by the upcoming Serenity (Eth2) upgrade [106]. This will likely make ZeeStar transactions highly affordable.

3.7 RELATED WORK

We now discuss previous work related to ZeeStar.

PAYMENT PRIVACY We already discussed works bringing privacy to cryptocurrency transactions in §1.1. Similar to ZeeStar, several of these works combine homomorphic encryption and NIZK proofs [29, 30, 31, 107].

In contrast to these works, ZeeStar brings data privacy to general applications beyond payments and combines NIZK proofs and homomorphic encryption automatically. ZeeStar can be used to implement private payments: for instance, it can readily express the confidential variant of Zether (see §3.6.1).

6 “Standard” gas price according to `https://www.gasnow.org/` (discontinued; accessed on 2021-07-23) for transactions expected to be mined within 3 minutes.

TABLE 3.3: Tools for smart contract privacy. We indicate which tools rely on minimal trust assumptions (🛡), support Ethereum (♦), can operate on foreign values (👥), or do not require the developer to instantiate cryptographic primitives (🔓).

Tool	🛡	♦	👥	🔓	**Remarks**
Arbitrum [33]	○	○	●	●	trusted manager
Ekiden [34]	○	○	●	●	trusted hardware
Hawk [32]	○	○	●	●	trusted manager
zkHawk [36]	◐[1]	○	●	●	requires interactive parties
smartFHE [38]	●	○	◐[2]	○	
ZEXE [39]	◕[3]	○	○	◐[4]	non-standard exec. model
Zether [29, 30]	●	●	◐[5]	◐[6]	limited applications
zkay (Chapter 2)	◕[3]	●	○	●	
ZeeStar (this chapter)	◕[3]	●	◕[7]	●	

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

SMART CONTRACT PRIVACY In Tab. 3.3, we provide a structured comparison of existing approaches for smart contract privacy.

As we already discussed in §2.10, Arbitrum [33], Ekiden [34], and Hawk [32] expose significant attack surface by relying on trusted managers or hardware. While zkHawk [36] replaces Hawk's manager by secure multi-party computation (SMC), it requires interactive parties.

The recent SmartFHE [38] system (published after the creation of zkay) proposes private smart contracts based on NIZK proofs and fully homomorphic encryption (FHE), where instantiating the latter at practical efficiency is known to require cryptographic expertise [108]. Further, its single-key variant requires all private inputs for a transaction to be owned by the same party. This is in contrast to ZeeStar, where the sender can combine self-owned and foreign values of multiple parties. Unfortunately, according to the authors, SmartFHE's multi-key variant is currently not practical.

ZEXE [39] targets a stronger privacy notion than ZeeStar by additionally hiding the involved parties and the executed function. Unlike ZeeStar, it requires the transaction sender to decrypt all input data and hence cannot operate on foreign values. Further, it uses a non-standard execution model based on records and predicates, and contract logic needs to be manually encoded as multiple R1CS for zk-SNARKs.

While Zether [29, 30] targets payments, it can be used for a limited set of wrapper applications that rely on its interface. However, any other applications or changes to the system (e.g., enforcing a minimal amount for transactions) require manually adapting the involved cryptographic primitives.

Compared to zkay (introduced in Chapter 2), ZeeStar comes with greater expressivity as it allows operations on foreign data.

To the best of our knowledge, ZeeStar is the first tool for general private smart contracts that allows operation on foreign values, automatically instantiates cryptographic primitives, and minimizes trust assumptions. Note that while our implementation currently uses zk-SNARKs with a trusted setup phase, ZeeStar's approach is proof-system-agnostic (see also §3.5.1).

HOMOMORPHIC ENCRYPTION AND ZK-SNARKS SAVER [109] proposes a "SNARK-friendly" homomorphic encryption scheme with efficient plaintext access within zk-SNARK proof circuits. While it is presented in the context of voting, it can be used in more general private smart contracts. ZeeStar leverages curve embedding for efficiency, but could likely integrate SAVER as an alternative.

3.8 SUMMARY

In this chapter, we presented ZeeStar, an extension of zkay by additively homomorphic encryption.

In contrast to zkay, ZeeStar allows computation on unknown data, enabling it to express key applications such as private payment systems and complex applications such as oblivious transfer. By relying on the language and privacy type system of zkay, ZeeStar can realize intuitive privacy specifications on permissionless blockchains, without requiring developers to manually instantiate cryptographic primitives.

Our end-to-end implementation of ZeeStar for Ethereum is practical and its gas costs are comparable to popular existing applications.

4 A SMART CONTRACT SYSTEM WITH DATA AND IDENTITY PRIVACY

The systems presented in Chapters 2–3 provide data privacy, which is useful to express many important applications. However, the presented systems do not hide the accessed memory locations and publish the identity of the transaction sender. Fundamentally, this prevents smart contracts from fully hiding all parties involved in a transaction—a notion we call *identity privacy* (see also Chapter 1). In this chapter, we explore how to achieve identity privacy for general smart contracts by designing a new smart contract system independent of zkay and ZeeStar.

4.1 INTRODUCTION

Identity privacy is concerned with hiding the parties involved in a transaction. For instance, the sender and recipient of a transferred coin should remain anonymous. Achieving identity privacy is challenging: merely removing account information from transactions is often not sufficient. First, transactions may perform data-dependent operations, where the accessed memory locations depend on information associated with accounts and may hence leak this information. Second, even if accounts are not publicly linked to physical users, transactions involving the same accounts may still be matched. For example, Bitcoin is notoriously vulnerable to deanonymization attacks, which identify the sender of individual transactions by tracing coins [110].

PRIVACY FOR CRYPTOCURRENCIES As we have discussed in §1.1, addressing the privacy issues of cryptocurrencies has been the focus of a long line of work [17, 18, 19, 20, 21, 23, 24, 26, 29, 30, 31].

Achieving privacy is challenging: systems based on mixers typically only provide weak privacy guarantees [22], and even cryptographic approaches such as Monero [23] are vulnerable to coin tracing attacks [40, 111].

Fortunately, some systems provide very strong privacy guarantees. For instance, Zerocash [25] (commercially deployed as Zcash [26]) cryptographically shields all sensitive aspects of its transactions. Specifically, it leverages an oblivious Merkle tree construction to hide the sender and receiver of a coin, as well as which coin was transferred.

PRIVACY FOR SMART CONTRACTS While Zerocash provides strong privacy guarantees, it is not programmable. In contrast, almost all programmable permissionless blockchains expose their users to deanonymization attacks or require strong trust assumptions (see §4.12 for details). The only exception is ZEXE [39], which reliably prevents deanonymization attacks. Conceptually, ZEXE extends Zerocash to programmable *records* (units of data generalizing the concept of a coin) produced and consumed by transactions. ZEXE inherits the strong privacy guarantees of Zerocash, protecting not only the sender's identity, but also the involved records, their data, and the logic itself.

SHORTCOMINGS OF ZEXE Unfortunately, ZEXE suffers from multiple shortcomings (see §4.11 for details). First, its applications are prone to vulnerabilities, to the point where even the original authors missed two attacks (presented in §4.11) on their motivating example. Further, the prevention of one attack as recommended by the authors requires the smart contract programmer to be intimately familiar with the combination of zero-knowledge proofs and key-private public-key cryptography. Second, ZEXE obstructs modular development of applications, as cooperating contracts must typically be aware of each other to prevent future, malicious contracts from bypassing the logic of existing contracts. Third, deploying a new application on ZEXE requires a setup performed by a trusted party to ensure the contract logic cannot be bypassed. Finally, ZEXE relies on a non-standard and low-level programming model in terms of *predicates*, which is unfamiliar to most developers.

ZAPPER: PRIVACY-PRESERVING SMART CONTRACTS In this chapter, we present *Zapper*, a novel privacy-focused smart contract system. Zapper allows developers to implement smart contracts in an intuitive, Python-embedded frontend with a standard high-level programming model. It provides data and identity privacy by hiding the involved parties, the data, and the objects (i.e., memory locations) accessed in a transaction. Additionally, Zapper achieves correctness (transactions respect the contract logic), access control (malicious contracts cannot bypass the logic of other contracts), integrity (transactions cannot be tampered with or replayed), and availability (valid transactions are not rejected), thus preventing the vulnerabilities found in ZEXE applications.

APPROACH Our key technical insight is to leverage a novel combination of an oblivious Merkle tree construction and a NIZK processor. Our oblivious Merkle tree construction hides the accessed objects and relies on techniques from Zerocash [25, 26] and ZEXE [39], adapted to our context and avoiding the vulnerabilities of ZEXE applications.

Our NIZK processor performs provably correct state updates without revealing private information, and is inspired by a separate line of work [82, 83, 84]. In contrast to ZEXE, using a NIZK processor avoids requiring a trusted party for deploying new contracts. To execute contracts on this processor, Zapper compiles them to a custom assembly format. Importantly, it sandboxes contracts by limiting their interactions to function calls. This access control facilitates modular development of contracts.

We note that selecting and combining these techniques to form a private, efficient, and secure smart contract system is a challenging task, as evidenced by the shortcomings of ZEXE (see also §4.11).

OUTLINE The remainder of this chapter is organized as follows.

- We start by providing an overview of Zapper (§4.2), after which we present the compilation of Zapper contracts to our custom assembly language (§4.3).
- In §4.4, we introduce the cryptographic components used in Zapper.
- Next, we discuss how the system state is updated by transactions (§4.5), and how Zapper ensures correct execution of contract logic (§4.6).
- In §4.7, we discuss the key security properties ensured by Zapper.
- We then present our end-to-end implementation of Zapper[1] (§4.8), followed by a thorough evaluation (§4.9) and a discussion of Zapper's limitations (§4.10).
- Finally, we discuss related work (§4.11–§4.12) and conclude the chapter (§4.13).

1 The source code is publicly available at `https://github.com/eth-sri/zapper`

(a) Coin class

```
class Coin(Contract):
  amount: Uint
  currency: Long
  # owner: Address (declared implicitly)

  @constructor
  @internal
  def create(self, amt: Uint, c: Long, o: Address):
    self.amount = amt; self.currency = c
    self.owner = o

  @constructor
  def mint(self, amt: Uint):
    self.amount = amt; self.currency = self.fresh()
    self.owner = self.me

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

  def split(self, amt: Uint) -> Coin:
    self.require(self.owner == self.me)
    self.require(self.amount >= amt)
    self.amount -= amt
    return self.new_obj(Coin.create, amt,
                        self.currency, self.me)
```

(b) DexOffer class

```
@has_address
class DexOffer(Contract):
  maker: Address; coin: Coin
  for_amount: Uint; for_currency: Long
  # owner: Address (declared implicitly)

  @constructor
  def create(self, shared: Address,
             coin: Coin, a: Uint, c: Long):
    self.maker = self.me; self.coin = coin
    self.for_amount = a
    self.for_currency = c
    self.owner = shared
    coin.transfer(self.address)

  def accept(self, other: Coin):
    self.require(
      other.amount   == self.for_amount and
      other.currency == self.for_currency
    )
    self.coin.transfer(
      self.me,
      sender_is_self=True
    )
    other.transfer(self.maker)
    self.kill()
```

FIGURE 4.1: Zapper classes modeling a coin (a) and a decentralized exchange (b), inspired by [39, §V].

4.2 OVERVIEW

We next provide an overview of Zapper.

4.2.1 *Running Examples: Coin and Exchange*

Fig. 4.1 shows implementations of a coin (Fig. 4.1a) and a decentralized coin exchange (DEX; Fig. 4.1b) in our Python-embedded frontend.

COIN A coin consists of an amount (Line 2), a currency (Line 3), and an owner identified by an address (Line 4, discussed shortly).

Users can issue a transaction to call a function of the coin. For example, the `transfer` function (Line 17) transfers the coin to a new owner by overwriting the `owner` field (Line 19). Here, Line 18 rejects (i.e., aborts and reverts) the transaction unless the *sender* (the address of the account used to create the transaction) is the coin's owner. The expression `self` indicates the *receiver object* of the transaction, while `self.me` holds the address of the sender.

Function `split` (Line 21) splits the coin into two coins while preserving the total amount. The function `merge` (omitted) merges two coins. Here, Line 24 decreases the amount of the original coin by `a`, while Line 25 creates a new coin with amount `a` via the constructor `create`. To prevent users from creating coins "out of thin air," the `create` function is marked as *internal* (Line 7), meaning that it can only be called from within the `Coin` class. The only non-internal constructor is `mint` (Line 13), which creates a *new* currency with a fixed total amount. This function leverages the built-in `fresh()` expression (Line 14) to obtain a fresh currency identifier. This identifier is guaranteed (with overwhelming probability) to be unique, preventing the minting of pre-existing currencies.

OWNERSHIP To ensure that Zapper objects are private, Zapper encrypts them under the public key of their owner, which is stored in a dedicated `owner` field implicitly available in every class (see Line 4). Specifically, `owner` holds the public key pk_α (serving as the *address*) of an account α. Only users with access to the corresponding secret key sk_α can read the contents of the object or interact with it.

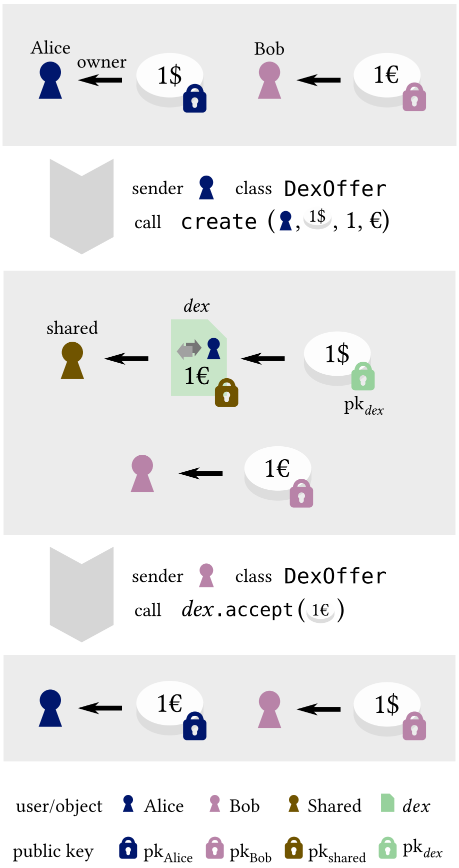


FIGURE 4.2: An example usage of the Zapper classes in Fig. 4.1.

DEXOFFER AND OBJECT OWNERSHIP Class `DexOffer` allows a `maker` to offer an exchange of a given `coin` (Line 29) for another coin of a specific amount and currency (Line 30).

Fig. 4.2 visualizes an example usage of `DexOffer`. Initially, Alice (●) and Bob (●) own one dollar (1$) and one euro (1€) coin, respectively. To offer a coin exchange, Alice first creates a shared user account (●) and distributes its key pair $(sk_{shared}, pk_{shared})$ to anyone she is willing to trade with, including Bob. Then, she creates a `DexOffer` object *dex* by calling `create` (Line 34), using her own account as the sender and passing (i) the public key of the shared account, (ii) her coin, and (iii) the expected amount of 1€ to be received in return.

To ensure that Alice cannot spend her coin while the offer still stands, Line 40 changes the coin's owner to *dex* by calling `transfer`. To this end, `DexOffer` is annotated as `@has_address`, which indicates that its instances are assigned their own *object account* and can therefore own other objects. Specifically, the address of *dex* is available via `self.address` (see Line 40). Note that two public keys are relevant to *dex*: while *dex* is owned by $\mathrm{pk}_{\mathrm{shared}}$ 🔒, the 1$ coin is transferred to *dex* using *dex*'s own key pk_{dex} 🔒 (middle of Fig. 4.2).

As `transfer` updates the owner of the 1$ coin to *dex*, Alice can no longer use her own account to spend it. However, the secret key sk_{dex} of *dex* is stored as part of *dex*, allowing Alice and Bob (who know $\mathrm{sk}_{\mathrm{shared}}$) to access it. To prevent both users from spending the coin, Zapper prohibits object accounts to be used as sender accounts for a transaction. Thus, once the coin is transferred to *dex*, the `require` statements in Line 18 and Line 22 of `Coin` prevent Alice and Bob from interacting with the coin directly, even though they know the secret key sk_{dex} of the coin's owner. Instead, Alice and Bob must interact with the coin via *dex* as follows.

To accept the offer, Bob calls `accept`, using his own account as the sender and passing his 1€ coin (checked in Lines 44–45). The function transfers the 1$ coin to Bob (Line 47) and Bob's 1€ coin to Alice (Line 51). By default, the sender address `self.me` is inherited in a nested function call: in Line 51, the call to `transfer` uses Bob as the sender. However, because the 1$ coin is owned by *dex*, the call in Line 47 sets the `sender_is_self` flag (a reserved argument implicitly defined for any function), which sets the sender address inside `transfer` to *dex*. Finally, Line 52 in `accept` destroys *dex* and makes it inaccessible to future transactions.

If Bob refuses to accept the offer, Alice can reclaim her 1$ coin owned by *dex* by calling `abort` (omitted). Further, to prevent unexpected privacy leaks, the owner of an object with its own address (`@has_address`) cannot be modified after construction (see §4.7).

DISCUSSION The decentralized exchange application implemented in Fig. 4.1 is inspired by [39, §V] and captures an important practical use case. In particular, `DexOffer` allows exchanging coins without handing over custody of coins to trusted centralized exchanges, which are notoriously vulnerable to attacks [112, 113]. Further, its privacy properties (discussed shortly) hide the most sensitive aspects of trading patterns (most importantly, the user identity and involved amounts), thus preventing attackers from exploiting these [114]. See [39] for a more elaborate discussion.

While in our example, the maker shares the offer details with all potential traders, the application can also be instantiated differently to provide more privacy. For example, the maker could share $\mathrm{sk}_{\mathrm{shared}}$ only with a centralized order book service which then connects potential trading partners. Further, the maker can remain anonymous by first creating an ephemeral user account and then transferring the offered coin to this account before creating *dex*. Once *dex* is accepted, the maker can privately transfer the received coin back to its original account.

4.2.2 *Security Properties Guaranteed by Zapper*

We now discuss the security properties ensured by Zapper.

PRIVACY Zapper ensures *identity privacy*: For every transaction, it hides the sender address (i.e., the value of `self.me`). This avoids revealing the `Coin` owner and hides the trading patterns of users.

Zapper also ensures *data privacy*: First, the values of all object fields are only visible to users knowing the owner's secret key. For example, only these users can see a coin's amount, and only users with access to $\mathrm{sk}_{\mathrm{shared}}$ can see the details of *dex*. Second, function arguments and return values of a transaction are only visible to the user creating the transaction. Importantly, this includes the identity of the receiver object (i.e., the value of the `self` argument). In our example, this ensures that coins cannot be tracked: it is hidden which of the coins in the system is modified by a `transfer` transaction. This is critical because revealing the receiver object enables tracing attacks which can compromise the sender's identity [40].

CORRECTNESS Zapper ensures that the logic defined in function bodies cannot be violated at runtime. Combined with access control (discussed next), this ensures that the behavior of a coin is completely defined by its implementation, even if used by untrusted users or code. For example, `Coin` ensures that the total amount of all coins of a specific `currency` remains constant after minting.

ACCESS CONTROL Zapper classes are subject to access control, which ensures that correctness cannot be violated by other, potentially malicious, classes. First, as discussed in §4.2.1, calls to internal functions are only permitted from within the same class. Second, the fields of an object can only be written from within a function of the same class, forcing any state

```
class Coin:
  amount: Uint
  currency: Long

  def transfer(self, arg: Address) -> None
    CID tmp0 self              # tmp0 ← class id of self
    EQ tmp1 tmp0 'Coin'        # tmp1 ← tmp0 == 'Coin'
    REQ tmp1                   # reject unless tmp1 == 1
    LOAD tmp2 self 'owner'     # tmp2 ← self.owner
    EQ tmp3 tmp2 me            # tmp3 ← tmp2 == me
    REQ tmp3                   # reject unless tmp3 == 1
    STORE arg self 'owner'     # self.owner ← arg

  ... # other functions
```

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

FIGURE 4.3: Zapper assembly language (Zasm) representation of `Coin`, using named registers for the sender (`me`), receiver (`self`), temporaries (`tmp`x), and arguments (`arg`).

changes to be performed via the function interface. For example, `DexOffer` cannot directly update the `owner` of a coin but must use the `transfer` function instead.

INTEGRITY AND AVAILABILITY Finally, Zapper ensures that transactions cannot be tampered with or replayed (integrity), and that valid transactions are not rejected (availability). In our example, availability ensures that the recipient of a coin is guaranteed to have access to it after `transfer` has been executed.

4.2.3 *Zapper Components*

We now discuss the two main components of Zapper. The *Zapper client* allows users to create classes and transactions. The *Zapper executor* stores classes and objects, and executes transactions.

LEDGER While users run the Zapper client on their local machine, the Zapper executor runs on top of a shared ledger, whose consensus mechanism maintains a globally consistent view of the system state. The realization of such a ledger is out of the scope of this work—Zapper is ledger-agnostic and could for instance be deployed on an extension of the Zcash [26] ledger, which maintains data structures similar to Zapper.

ASSEMBLY CODE Before submitting a new Zapper class to the Zapper executor, the Zapper client compiles it to a custom Zapper assembly language (Zasm) format. Zasm code consists of instructions for a (virtual) processor ran when executing a transaction.

For example, Fig. 4.3 shows the Zasm code of `Coin`, focusing on function `transfer`. Lines 6–8 perform a type check of `self`. This step is necessary to prevent malicious users or code from passing a receiver of non-`Coin` type to `transfer` and thereby bypassing Zapper's access control. Here, **`CID`** loads the *class id* (a number identifying the class) of `self` into a temporary register `tmp0`. The **`EQ`** instruction stores 1 into `tmp1` if and only if the two arguments are equal (0 otherwise), where `'Coin'` represents the class id of `Coin` in a readable fashion. Next, Lines 9–11 ensure that only the coin owner can transfer coins (see Line 18 in Fig. 4.1a), where `'owner'` represents the numerical offset of field `owner` in `Coin`. Finally, Line 12 updates the coin's owner.

ASSEMBLY STORAGE The Zapper executor stores Zasm code in an *assembly storage*. It enforces access control by inferring the type of registers in Zasm code and then checking whether the code respects the relevant access control policies. For instance, Zapper checks that internal functions are only called from within the same class and that only fields of the same class as `self` are written by **`STORE`**.

In order to verify whether potentially untrusted Zasm code matches a trusted Zapper Python class, users can separately compile the class and check equality of Zasm code (analogously to how EVM bytecode is checked to match Solidity code in Ethereum).

SYSTEM STATE Fig. 4.4a visualizes the system state maintained by the Zapper executor. In particular, objects are stored in an *object tree* T. More precisely, the data of an object is encrypted under the owner's public key to obtain a *record*. The records representing the current and past states of any object in the system are stored as leaves in T, which is an append-only Merkle hash tree [49] whose root β is a cryptographic summary of the object states.

Zapper further maintains two auxiliary data structures. First, to invalidate records accessed by transactions (see shortly), Zapper uses an append-only

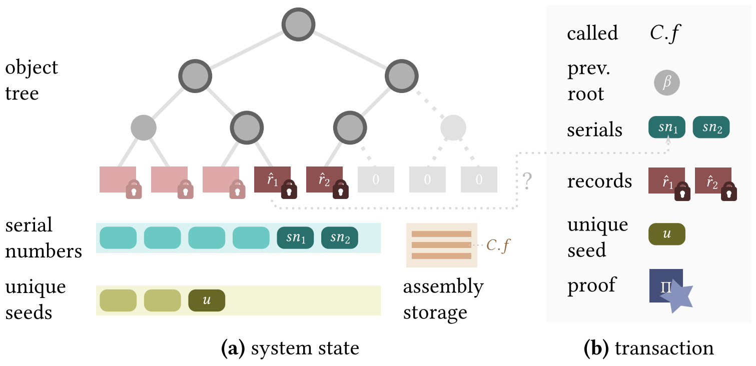


FIGURE 4.4: State maintained by the Zapper executor. New transactions insert the darker rectangles and update the rounded circles.

list of unique *serial numbers*. Second, an append-only list of *unique seeds* allows various components of Zapper to produce provably unique values. For instance, these seeds are used to compute values returned by `fresh()` (Line 14 in Fig. 4.1a).

TRANSACTIONS The Zapper client allows users to execute a function f of a previously registered Zasm class C by sending a transaction to the Zapper executor (see Fig. 4.4b). Conceptually, this transaction executes the Zasm instructions of $C.f$ on the Zapper processor and updates the state of the involved objects by inserting new records into the object tree. To invalidate the previous state of the objects accessed (by reads or writes) in the transaction, the transaction includes a list of serial numbers which uniquely but privately indicate the accessed records. Enforcing the uniqueness of serial numbers then ensures that each record is accessed at most once. Similarly, a unique seed is produced uniformly at random.

Importantly, the unique seed, serial numbers, and new records do not leak any information about the data, objects, or users involved in the transaction, thus maintaining both data and identity privacy.

ENSURING CORRECTNESS The user also includes a NIZK proof Π in the transaction to certify that the new records and serial numbers were computed correctly. This proof is verified by the Zapper executor, which upon success inserts the records, serial numbers, and unique seed into the object tree, serial number list, and unique seed list, respectively (Fig. 4.4).

4.3 ASSEMBLY CODE

Next, we provide details on the Zasm code generated by the Zapper client (§4.3.1). Then, we discuss how this code is processed by the Zapper executor before storing it in the assembly storage (§4.3.2).

Overall, Zasm allows to enforce access control via type checking and is designed to enable efficient execution on a NIZK processor.

4.3.1 *Assembly Code in Client*

The Zapper client compiles classes expressed in Zapper's Python frontend to Zasm code. As this step is conceptually straightforward, we only discuss the resulting Zasm code.

TYPES AND VALUES Zasm code contains type information. Supported types include unsigned integers (**`Uint`**), addresses (**`Address`**), and pointers to objects. For technical reasons (see §4.6.2 in §4.6), values produced by `fresh()` are of the special large unsigned integer type **`Long`** precluded from arithmetic operations.

All values are elements of a prime field $\mathbb{F}_q = \{0, \ldots, q-1\}$ for a large prime number q, allowing for efficient correctness checks of processor execution (see §4.8). Pointers to an object hold the object's *object id*—a unique identifier in $\mathbb{F}_q$ assigned to each object.

CLASSES AND INSTRUCTIONS The Zasm code of a class defines its fields and functions, including the types of fields, function arguments, and return values (see Fig. 4.3). Further, function bodies are represented by a sequence of Zasm instructions. In Tab. 4.1, we list all Zasm instructions. The instruction set has been specifically designed to allow for efficient generation of NIZK correctness proofs (see §4.6).

REGISTERS Zasm code operates on named registers `reg`, each holding a value in $\mathbb{F}_q$. The sender address (`self.me` in Zapper's frontend), and function arguments are available in dedicated registers.

BASIC OPERATIONS Zasm supports standard arithmetic $(+, -, \cdot)$, comparison $(<, =)$, and conditional assignment operations (assigning some value to a register iff a condition is true). Comparison operations return a value 0 or 1 interpreted as false or true, respectively. Zapper can express

TABLE 4.1: The Zasm instruction set, grouped by generic (upper part) and Zapper-specific (lower part) instructions. Arguments marked as $\langle x \rangle$ may be both constants or references to a register. We use $\text{val}(x)$ for the value of the constant or register indicated by $\langle x \rangle$.

NOOP	No operation
MOV $d\ \langle s \rangle$	Writes $\text{val}(s)$ to `reg[`d`]`
CMOV $d\ \langle c \rangle\ \langle s \rangle$	Writes $\text{val}(s)$ to `reg[`d`]` if $\text{val}(c) = 1$
ADD $d\ \langle s_1 \rangle\ \langle s_2 \rangle$	Writes $\text{val}(s_1) + \text{val}(s_2)$ to `reg[`d`]`
SUB $d\ \langle s_1 \rangle\ \langle s_2 \rangle$	Writes $\text{val}(s_1) - \text{val}(s_2)$ to `reg[`d`]`
MUL $d\ \langle s_1 \rangle\ \langle s_2 \rangle$	Writes $\text{val}(s_1) \cdot \text{val}(s_2)$ to `reg[`d`]`
EQ $d\ \langle s_1 \rangle\ \langle s_2 \rangle$	Writes 1 to `reg[`d`]` if $\text{val}(s_1) = \text{val}(s_2)$, 0 otherwise
LT $d\ \langle s_1 \rangle\ \langle s_2 \rangle$	Writes 1 to `reg[`d`]` if $\text{val}(s_1) < \text{val}(s_2)$, 0 otherwise
REQ $\langle c \rangle$	Aborts transaction if $\text{val}(c) \neq 1$
LOAD $d\ \langle oid \rangle\ i$	Loads i-th field of object with id $\text{val}(oid)$ into `reg[`d`]`
STORE $s\ \langle oid \rangle\ i$	Stores `reg[`s`]` into the i-th field of object with id $\text{val}(oid)$
CID $d\ \langle oid \rangle$	Writes the class id of object with id *oid* to `reg[`d`]`
PK $d\ \langle oid \rangle$	Writes the public key of object with id *oid* to `reg[`d`]`
NEW d *cid*	Creates a new object with class id *cid* and writes its object id to `reg[`d`]`
KILL $\langle oid \rangle$	Destroys the object with id $\text{val}(oid)$
FRESH d	Writes a unique secret value to `reg[`d`]`
NOW d	Writes the current timestamp to `reg[`d`]`

boolean operations by arithmetic operations (e.g., a && $b = a \cdot b$). Operation **REQ** enforces assertions.

LOADING AND STORING OBJECT DATA Zasm provides object-aware memory instructions to access objects by their object id. While this prohibits advanced techniques such as pointer arithmetic, it enables efficient access of object data within NIZK proof circuits (see §4.6).

For instance, the **LOAD** instruction first finds the object with object id $\text{val}(oid)$ and then loads the i-th field into the target register `reg[`d`]` (during compilation, each field of a class is assigned a numerical offset). The **STORE** instruction works analogously. Zasm further allows accessing object metadata. First, the **CID** instruction gets the class id of an object. This is useful to realize runtime type checks: for example, Lines 6–8 in Fig. 4.3 ensure that `self` is a `Coin`. Second, **PK** returns the object's own address (public key) if it has one.

CREATING AND DESTROYING OBJECTS The **`NEW`** instruction stores the object id of a new object of a given class into a target register. Subsequent **`STORE`** instructions can then be used to populate the fields of the new object. Conversely, the **`KILL`** instruction destroys a given object, making it inaccessible for future instructions.

FRESH VALUES AND TIMESTAMPS The **`FRESH`** instruction creates and stores a unique secret value into a target register (see `fresh()`).

The Zapper executor maintains a clock at coarse granularity. The **`NOW`** instruction stores the current timestamp into a target register.

FUNCTION CALLS Function calls are represented by a special **`CALL`** instruction indicating (i) the called function, (ii) the `sender_is_self` flag determining whether the sender is set to `self.address` or inherited, and (iii) the arguments, including the receiver object id. The **`CALL`** instruction is not a native instruction supported by the Zapper processor and is hence not shown in Tab. 4.1. Instead, as we will see later, function calls are inlined before execution.

CONTROL FLOW To allow for efficient generation of NIZK correctness proofs (§4.6), Zasm code does not support control flow, jumps, loops, or operations modifying Zasm instructions at runtime (i.e., we assume a Harvard architecture). In particular, Zasm instructions are always executed in the given order.

However, by representing if-then-else using conditional assignments and statically unrolling loops up to an upper bound, most smart contracts can be expressed in Zasm.

4.3.2 *Assembly Code in Executor*

When the Zapper client registers a new Zasm class at the Zapper executor, the latter ensures it does not violate access policies and prepares it for execution on the Zapper processor.

MALICIOUS CODE Zasm code received at the Zapper executor cannot be trusted, since an attacker could try to craft malicious Zasm code bypassing the logic specified in existing Zasm classes. For instance, an attacker could use **`STORE`** to increase the `amount` of a `Coin` object from a different class. To prevent such attacks, Zapper enforces multiple access policies, as discussed next.

ACCESS CONTROL Zapper enforces several access policies by default. First, **`STORE`** instructions within a class C can only target objects of type C, ensuring that fields of other classes cannot be written directly. Similarly, to prevent the destruction of unrelated objects, **`KILL`** may only destroy objects of type C. Further, **`NEW`** may only create objects of type C (objects of different type C' must be created via a **`CALL`** to a constructor of C'). Also, **`STORE`** cannot be used to update the owner field of classes annotated as `@has_address` outside constructors (we discuss the necessity of this rule in §4.7). Finally, the dedicated register holding the sender address (authenticated by Zapper) must not be overwritten by any instruction.

We note that by design, Zapper ensures that the user creating a transaction knows the secret key of the owner of any accessed object (by **`LOAD`**, **`STORE`**, **`CID`**, **`PK`**, or **`KILL`**), as their records must be decrypted. This is *not* equivalent to `self.require(self.owner == self.me)`: a user can only use one sender account (which must be a user account) but may have access to the secret keys of many accounts (including object accounts). For example, in the second transaction of Fig. 4.2, Bob uses $\text{sk}_{\text{shared}}$, sk_{dex}, and sk_{Bob} to decrypt the inputs, but pk_{Bob} as the sender address.

In addition to the default access policies, Zapper allows specifying custom policies for functions. In particular, the annotation `@only(`C`)` declares a function *internal to* a class C, making it inaccessible to any class $C' \neq C$. Zapper also provides an annotation `@internal` as a shorthand for `@only(`C`)`, where C is the current class.

STATIC CHECKS To enforce the above policies, the Zapper executor performs a static type analysis on the received Zasm code. First, it checks whether all fields of new objects are initialized (uninitialized objects can violate type safety). Second, it checks whether the code is well-typed (e.g., arithmetic operations are only performed on `Uint` values, the types of arguments in **`CALL`** match the target's function signature, etc.) and determines the target class of each **`LOAD`**, **`STORE`**, and **`CALL`** instruction. Next, Zapper checks whether the code satisfies all default and custom access policies described above. If any of these checks fail, the Zasm code is rejected.

RUNTIME CHECKS The types of any pointer arguments cannot be checked statically as their value is selected by a potentially malicious user at runtime. Hence, the Zapper executor inserts dynamic type checks for these arguments, relying on the **`CID`** instruction.

For example, consider the `transfer` function of `Coin` (Fig. 4.1a). To ensure that the first argument `self` is of the expected type `Coin`, Zapper inserts Lines 6–8 in Fig. 4.3.

INLINING CALLS The Zapper processor does not natively support **`CALL`** instructions. Instead, the Zapper executor inlines the function body of any called function. Here, the sender address register of the called function is correctly inherited or initialized with the caller address, depending on the `sender_is_self` flag. For inlining to succeed, Zapper disallows (mutually) recursive function calls.

ALLOCATING REGISTERS The Zapper processor only supports a fixed number N_{regs} of registers $\texttt{reg[0]}, \ldots, \texttt{reg[}N_{\text{regs}} - 1\texttt{]}$ for a system parameter N_{regs}. Hence, in a final step, the Zapper executor allocates all named registers to these indexed registers. Then, the Zasm code is stored in the assembly storage, ready to be used by transactions.

4.4 CRYPTOGRAPHIC COMPONENTS

We next discuss the cryptographic components used in Zapper.

ENCRYPTION To encrypt records, Zapper uses a *key-private* [115] asymmetric encryption scheme. This ensures that attackers cannot determine which public key was used to produce a given ciphertext, thus hiding the identity of an encrypted record's owner. We write $\text{Enc}(p, \text{pk}_\alpha, R)$ to denote the encryption of plaintext p under key pk_α with encryption randomness R.

MERKLE TREE Zapper relies on a Merkle hash tree [49] (see §1.3), which uses a collision-resistant hash function H to derive a root hash β of all records stored as leaves in the object tree T. This allows proving that a given record $\hat{r}$ is in T with root β using a *Merkle certificate* π [50, §2.1.1], and updating β upon insertion of new records.

HASH FUNCTIONS Zapper relies on a family of cryptographic hash functions $H_i \colon \{0,1\}^* \to \{0,1\}^{\Omega(\lambda)}$ with security parameter λ and the following properties: (i) H_i is collision-resistant, and (ii) the function $f \colon \{0,1\}^* \times \{0,1\}^{\Omega(\lambda)} \to \{0,1\}^{\Omega(\lambda)}$ defined as $f(x,k) := H_i(x \,\|\, k)$ is a pseudorandom function (i.e., for uniformly random k, the function $f(\,\cdot\,, k)$ is indistinguishable from a random function).

Various components of Zapper rely on H_i to derive a unique secret value z using the following generic construction, inspired by ZEXE [39] and Zcash [26]. For unique U and private R,

$$z = H_i(U \parallel R). \tag{4.1}$$

Computed as in Eq. (4.1), z has two key properties:

- *Secrecy:* For uniformly random R, any user not knowing R cannot distinguish z from a uniformly random value. This follows from pseudorandomness.
- *Uniqueness:* with overwhelming probability, z is unique, even for adversarially chosen R. This follows from collision-resistance.

NIZK PROOFS Recall that for a predicate ϕ and a public input x, a NIZK proof [41, 42] allows a prover to demonstrate that it knows a private input w s.t. $\phi(x; w)$ holds, without leaking any information about w beyond the fact that ϕ holds (see §1.3). In the context of Zapper, x includes the executed function body and information on the state before and after execution, while w includes private information known to the sender (e.g., secret keys), allowing ϕ to check that the function was executed correctly.

To ensure correctness, Zapper relies on an SE-SNARK [44]: a zk-SNARK satisfying perfect zero-knowledge, perfect completeness, and simulation-extractability [44].

4.5 TRANSACTIONS

We now discuss how the Zapper system state is represented and updated by transactions. Most importantly, our approach hides the accessed objects by an oblivious Merkle tree construction. To this end, we rely on techniques from Zerocash [25, 26] and ZEXE [39] but adapt them to our context and avoid the vulnerabilities of ZEXE.

4.5.1 *Example Transaction*

To provide some intuition, we first discuss an example transaction calling the `accept` function of `DexOffer` (Fig. 4.1b).

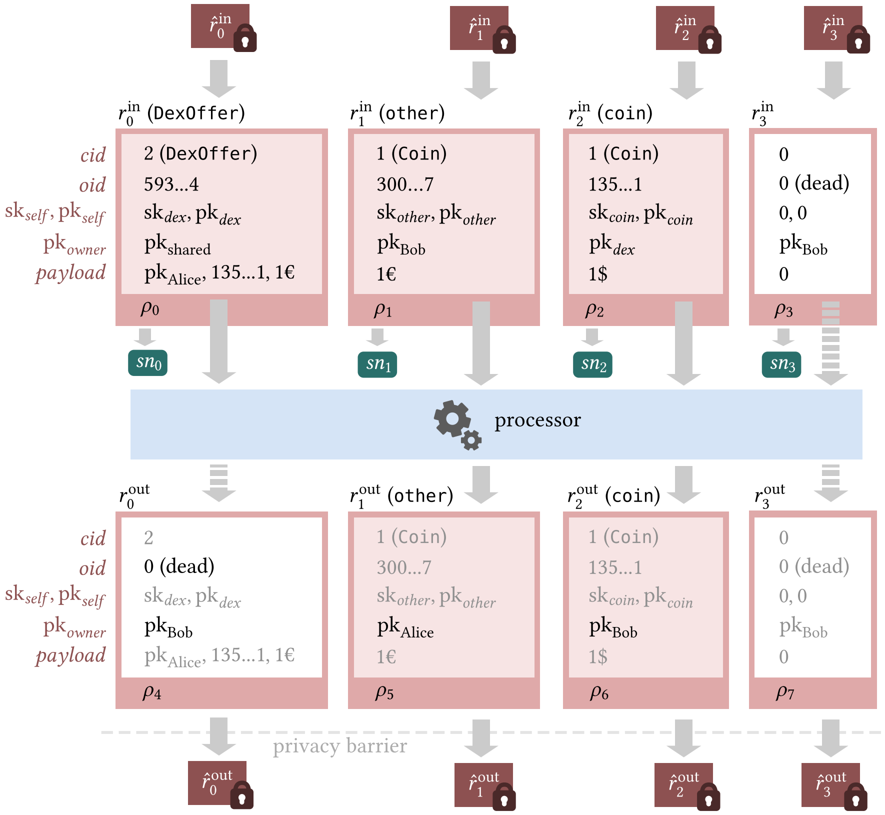


FIGURE 4.5: Visualization of a transaction for `DexOffer.accept` (Fig. 4.2).

INPUT RECORDS The `accept` function accesses several objects: (i) the `DexOffer` object `self`, (ii) the coin `other` transferred to the `maker`; and (iii) the coin `coin` transferred to the transaction sender.

Fig. 4.5 shows how `accept` modifies these objects when called by Bob (sender $\mathrm{pk}_{\mathrm{Bob}}$) with arguments $args = (593...4, 300...7)$ indicating the object ids of `self` and `other` (see the second transaction in Fig. 4.2). At a high level, Zapper first loads the *encrypted records* $\hat{r}_0^{\mathrm{in}}, \ldots, \hat{r}_2^{\mathrm{in}}$ of the accessed objects, which includes $\hat{r}_2^{\mathrm{in}}$ containing the state of `coin`, from the object tree (we discuss $\hat{r}_3^{\mathrm{in}}$ shortly). Next, these are decrypted to obtain *plain records* r_i^{in}.

PLAIN RECORDS Specifically, the *plain record* of an object is a tuple

$$r = (cid, oid, \mathrm{sk}_{self}, \mathrm{pk}_{self}, \mathrm{pk}_{owner}, payload),$$

where *cid* and *oid* are the class id and object id, respectively, pk_{owner} is the public key of the owner, and *payload* holds the values of the remaining fields. Further, $(\mathrm{sk}_{self}, \mathrm{pk}_{self})$ is the key pair of the object account. For simplicity, we also assign an account to objects not annotated as `@has_address`, however, this account is never used. Note that in general, pk_{self} (belonging to the object) and pk_{owner} (belonging to the object's owner) are keys of different accounts.

ENCRYPTED RECORDS Before encryption, a plain record r is extended by a secret *serial nonce* ρ (a globally unique number later used to invalidate outdated records). Then, it is encrypted under the public key of the object's owner to yield the *encrypted record*

$$\hat{r} = \mathrm{Enc}((r, \rho), r.\mathrm{pk}_{owner}, R),$$

where R is some encryption randomness. Note that in contrast to ZEXE, Zapper relies on encryption instead of commitments, preventing the denial-of-funds attack in ZEXE (see §4.11).

PROCESSOR After decryption, the plain records are fed to the Zapper processor, which executes the Zasm code of `accept` to produce the plain output records r_i^{out} (Fig. 4.5 highlights changed entries).

OUTPUT RECORDS Finally, the plain records r_i^{out} are extended by fresh serial nonces ρ_i and encrypted under the owners' public keys to obtain $\hat{r}_i^{\mathrm{out}}$.

These records will be published to the ledger and inserted into the object tree once the transaction is accepted.

DEAD RECORDS When an object is destroyed by **KILL**, the processor sets the object id *oid* of the corresponding record to the reserved value 0, indicating that this record is *dead*. Further, pk_{owner} of the record is set to the sender's public key such that the resulting record is hidden from other users. This is important as dead records may include stale data of the previously represented object.

In Fig. 4.5, the `DexOffer` object $\hat{r}_0^{\text{in}}$ is destroyed, hence *oid* of r_0^{out} is set to 0 and pk_{owner} is set to pk_{Bob}. The payload component of r_0^{out} still contains private data (e.g., the value of the `for_amount` field), but as it will be encrypted for Bob, no other user learns this.

The processor accepts exactly N_{obj} plain records as inputs, for a system parameter N_{obj}. The inputs are appropriately padded using artificial dead records. Similarly, the processor returns N_{obj} output records. In Fig. 4.5, it is $N_{\text{obj}} = 4$ and $\hat{r}_3^{\text{in}}$ is used as a padding record.

TRANSACTION CONTENTS Importantly, the steps visualized in Fig. 4.5 are hidden from the Zapper executor to maintain data and identity privacy (see the indicated privacy barrier). The data published by the Zapper client only consists of the encrypted output records $[\hat{r}^{\text{out}}]$ (bottom row in Fig. 4.5; we use the notation $[\cdot]$ to indicate a list) and some bookkeeping data (see also Fig. 4.4b).

Formally, a transaction $tx = (C.f, \beta, [sn], [\hat{r}^{\text{out}}], u, \Pi)$ consists of the fully qualified function name $C.f$ of the called function, the current root hash β of T, the serial numbers $[sn]$ of accessed records, a list $[\hat{r}^{\text{out}}]$ of new records, a unique seed u, and a NIZK proof Π certifying correctness. §4.5.2–§4.5.3 explain the purpose of these items.

4.5.2 *Creating Transactions*

Next, we describe how the Zapper client creates a transaction *tx* for a user who wishes to call a function $C.f$ with arguments *args*.

SIMULATION The Zapper client first loads the Zasm code *zasm* for $C.f$ from the assembly storage. It then locally simulates the execution of *zasm* with arguments *args*. During simulation, the most recent records of any pre-existing objects accessed by *zasm* (due to **LOAD**, **STORE**, **CID**, **PK**, or **KILL**)

Algorithm 4.1 The main NIZK proof circuit ϕ.

```
1:  **Public inputs:**
2:      $C.f$, $\beta$, $[sn]$, $[\hat{r}^{\text{out}}]$, $u$ as in Fig. 4.4b, timestamp $t$, code $zasm$
3:  **Private inputs:**
4:      $C'.f'$: called class and function id          $[r^{\text{out}}]$: plain output records
5:      $\text{sk}_{me}, \text{pk}_{me}$: key pair of sender          $[\pi]$: input record Merkle certificates
6:      $args$: arguments for call                      $[\text{sk}_\alpha]$: input record secret keys
7:      $[\hat{r}^{\text{in}}]$: encrypted input records            $[R], [R^{\text{pr}}]$: randomness
8:  Authenticate sender: $\text{pk}_{me} \overset{!}{=} \text{derivePk}(\text{sk}_{me})$ **and** $\text{isUser}(\text{pk}_{me}) \overset{!}{=} \text{true}$
9:  Check function: $C.f \overset{!}{=} C'.f'$
10: **for** $i \in \{0, \dots, N_{\text{obj}} - 1\}$ **do**
11:     Decrypt record: $(r_i^{\text{in}}, \rho_i^{\text{in}}) \leftarrow \text{Dec}(\hat{r}_i^{\text{in}}, \text{sk}_{\alpha_i})$
12:     **if** $r_i^{\text{in}}.oid \neq 0$ **then**
13:         Check that $\hat{r}_i^{\text{in}}$ is in Merkle tree with root $\beta$ (using $\pi_i$)
14:     **else**
15:         Check serial nonce: $\rho_i^{\text{in}} \overset{!}{=} H_2(i + N_{\text{obj}} \parallel u \parallel R_{i+N_{\text{obj}}})$
16:     Check serial number: $sn_i \overset{!}{=} H_1(\rho_i^{\text{in}} \parallel \text{sk}_{\alpha_i})$
17: Run processor (Alg. 4.2) for $zasm, u, t, \text{pk}_{me}, args, [r^{\text{in}}], [r^{\text{out}}], [R^{\text{pr}}]$
18: **for** $i \in \{0, \dots, N_{\text{obj}} - 1\}$ **do**
19:     Compute serial nonce: $\rho_i^{\text{out}} \leftarrow H_2(i \parallel u \parallel R_i)$
20:     Check encryption: $\hat{r}_i^{\text{out}} \overset{!}{=} \text{Enc}((r_i^{\text{out}}, \rho_i^{\text{out}}), r_i^{\text{out}}.\text{pk}_{owner}, R_{i+2N_{\text{obj}}})$
```

are fetched from a local copy of the object tree, collected in a list $[\hat{r}^{\text{in}}]$, and decrypted using the owners' secret keys. As a result, Zapper obtains (i) the return value of $C.f$, which is returned to the user; and (ii) the list $[r^{\text{out}}]$ of plain output records, which represent the updated states of the objects in $[\hat{r}^{\text{in}}]$ and any new objects. The list $[\hat{r}^{\text{in}}]$ is then padded by an appropriate number of dead records.

CORRECTNESS PROOF CIRCUIT To prove that $C.f$ was executed correctly, the Zapper client creates a NIZK proof Π.

Alg. 4.1 shows the proof circuit ϕ for Π. Conceptually, the public inputs of ϕ (Lines 1–2) are provided by the Zapper executor when verifying Π. In contrast, the private inputs of ϕ (Lines 3–7) are provided by the Zapper client when creating Π. These inputs include private information such as the keys $(\text{sk}_{me}, \text{pk}_{me})$ of the sender.

SENDER AUTHENTICATION To ensure that the sender cannot be impersonated by an unauthorized user, we check that the user creating Π knows the secret key of the sender account. More precisely, Line 8 uses derivePk to check whether sk_{me} corresponds to pk_{me}.

CHECKING THE FUNCTION Both the public and private inputs of ϕ include identifiers of the called class C and function f. By checking that these match (Line 9), the proof Π acts as a signature on $C.f$, ensuring that $C.f$ cannot be changed once the proof is generated.

ENFORCING OBJECT OWNERSHIP Zapper objects can be owned by other objects. For instance, during the lifetime of *dex* in Fig. 4.1, the coin `coin` is owned by *dex*. The `self.require(self.owner == self.me)` statements in `Coin` ensure that only *dex* can call its functions. Unfortunately, reflecting these requirements in ϕ by checking that pk_{me} equals the coin's owner address is not sufficient to prevent users from directly calling the functions of a coin owned by *dex*: the secret key sk_{dex} is stored as part of *dex*'s record and users with access to *dex* could hence use $(\mathrm{sk}_{dex}, \mathrm{pk}_{dex})$ as sender account key pair.

To prevent such object impersonation attacks, the public keys in the system are partitioned into keys of user and object accounts. Line 8 explicitly checks that pk_{me} belongs to a user account. In our implementation (§4.8), isUser returns the key's least significant bit.

ACCESSING OBJECTS Lines 11–13 access the input records $[\hat{r}^{\mathrm{in}}]$ while hiding their location in T. First, Line 11 checks that $[\hat{r}^{\mathrm{in}}]$ are correctly decrypted, yielding both serial numbers $[\rho^{\mathrm{in}}]$ (discussed shortly) and plain records $[\hat{r}^{\mathrm{in}}]$. Next, for each non-dead record $\hat{r}^{\mathrm{in}}_i$, Line 13 verifies a Merkle certificate π_i (passed as a private input to ϕ) showing that $\hat{r}^{\mathrm{in}}_i$ is a leaf of the current object tree with root β.

PREVENTING ACCESS OF OUTDATED STATE Recall that new records are appended to the object tree without replacing their original version (note that replacement would leak the accessed object). Hence, the construction presented so far only ensures that the accessed records represent *some* previous state of the respective objects, but not necessarily the most recent one. Therefore, we need to privately invalidate outdated records. To this end, Zapper relies on a technique introduced in Zerocash [25], which uses serial numbers to privately identify and invalidate accessed records.

In particular, to invalidate the accessed records $\hat{r}_i^{\text{in}}$, Zapper derives and publishes a unique *serial number* for each $\hat{r}_i^{\text{in}}$ as

$$sn_i = H_1(\rho_i^{\text{in}} \parallel \text{sk}_{\alpha_i}). \tag{4.2}$$

Here, ρ_i^{in} is the serial nonce contained in $\hat{r}_i^{\text{in}}$ and sk_{α_i} is the owner's secret key used to decrypt $\hat{r}_i^{\text{in}}$. Eq. (4.2) is checked in Line 16.

Because the computation of the serial number (Line 16) follows the generic construction in Eq. (4.1), assuming the serial nonces ρ_i^{in} are globally unique (discussed shortly), serial numbers of different records cannot collide (*uniqueness*). Further, the serial number sn_i is indistinguishable from a uniformly random value for any user not knowing sk_{α_i} (*secrecy*). This is important to ensure privacy: otherwise, a malicious user Eve who created $\hat{r}_i^{\text{in}}$ for owner Alice could tell when Alice accesses it by watching for sn_i.

Overall, the Zapper client computes the serial numbers $[sn]$ and includes them in *tx*. The Zapper executor will ensure that these are globally unique, enforcing that any record can be accessed at most once. As this also applies to records whose state is not changed but only read (e.g., by **LOAD**), the Zapper processor output includes plain records of objects which were only read. This ensures that a fresh record representing these objects is re-inserted into the object tree.

PROCESSOR EXECUTION Line 17 ensures that running the program *zasm* with arguments *args* and current timestamp t on inputs $[r^{\text{in}}]$ results in the plain output records $[r^{\text{out}}]$. As this step is more involved, we discuss it separately in §4.6.

DERIVING NEW SERIAL NONCES Before encrypting r_i^{out}, Zapper derives a new serial nonce ρ_i^{out} for r_i^{out} as in Line 19, repeated here:

$$\rho_i^{\text{out}} = H_2(i \parallel u \parallel R_i). \tag{4.3}$$

Here, u is a public and globally unique seed, and R_i is fresh randomness. Similarly, the serial nonce ρ_i^{in} of any dead *input* padding record is computed as in Line 15, repeated here:

$$\rho_i^{\text{in}} = H_2(i + N_{\text{obj}} \parallel u \parallel R_{i+N_{\text{obj}}}). \tag{4.4}$$

The seed u is selected uniformly at random by the Zapper client and included in the transaction *tx*. Like the serial numbers, u is enforced to be globally unique by the Zapper executor (note that for a large seed space, the selected u is unique with overwhelming probability). As we will see in §4.6, the seed u is also used by the processor to derive other unique values. Due to the uniqueness property of the construction in Eq. (4.1), the nonces computed by Eqs. (4.3)–(4.4) are globally unique with overwhelming probability.

In ZEXE, the serial numbers of dead inputs (called "dummy" by the authors) are selected freely by the user, allowing a "lock-out" attack on applications with shared keys (see §4.11). In contrast, Zapper prevents this attack using the construction in Eq. (4.4).

ENCRYPTION AND PROOF GENERATION Finally, Zapper encrypts the plain records $[r^{\text{out}}]$ along with their serial nonces using the respective owners' public keys to obtain the output records $[\hat{r}^{\text{out}}]$ (see Line 20). To complete the data in tx, the Zapper client generates a NIZK proof Π for ϕ. The transaction tx is then sent to the ledger.

4.5.3 *Processing Transactions*

We next describe how the Zapper executor processes transactions.

VALIDITY CHECKS When the Zapper executor receives a transaction $tx = (C.f, \beta, [sn], [\hat{r}^{\text{out}}], u, \Pi)$, it first looks up the assembly code $zasm$ for the called function $C.f$ in the assembly storage and prepares the public inputs $C.f, \beta, [sn], [\hat{r}^{\text{out}}], u, t, zasm$ for the proof circuit ϕ, where t is the current timestamp. Then, the ledger

(i) checks the validity of the proof Π;

(ii) checks if β is a valid previous root hash of T;

(iii) checks if the serial numbers in $[sn]$ are unique and do not already occur in the serial number list; and

(iv) checks if u does not already occur in the unique seed list.

STATE UPDATES If any of the validity checks fail, the transaction is rejected. Otherwise, the system state is updated as follows:

1. For all $sn_i \in [sn]$, insert sn_i into the serial number list.

2. Insert u into the unique seed list.

3. For all $\hat{r}_i^{\text{out}} \in [\hat{r}^{\text{out}}]$, insert $\hat{r}_i^{\text{out}}$ into the object tree T.

CONCURRENT TRANSACTIONS Note that another transaction tx' may have been accepted since the Zapper client started creating tx. For this reason, in the validity check (ii) above, β is not required to be the most recent root hash of T, but may be any previous root hash. As long as the unique seed and records accessed in tx are distinct from the unique seed and records accessed in any previous tx', their ordering does not matter and tx remains valid. In other words, concurrent transactions accessing *distinct* objects will not affect each other. Importantly, a third party which does not have access to (i.e., does not know the owner secret keys of) any object involved in another transaction tx cannot perform a front-running attack and block tx by trying to consume the same object(s).

If two concurrent transactions access the same records (i.e., read or write the same object), the ledger rejects the transaction tx it receives last. Assuming all assertions still hold and the involved objects still exist, the user creating tx can always re-create it.

We assume that the granularity of timestamps is coarse enough (e.g., in the order of hours) to account for the delay between transaction generation and verification. In case the current timestamp changes in-between, the user must re-create the transaction.

4.6 PROVING PROCESSOR EXECUTION

We now discuss our zero-knowledge processor, which allows the Zapper client to provably execute Zasm code. Our processor is inspired by previous work [82, 84], but adapted to reduce the cost of the most expensive operations: we prefetch all accessed objects (see also §4.5) and precompute the result of cryptographic operations.

4.6.1 *Emulating the Processor*

In Alg. 4.2, we show the sub-circuit checking correct execution of the provided Zasm code. This sub-circuit is part of the main proof circuit ϕ (Alg. 4.1) and emulates the cycles of the Zapper processor.

PRECOMPUTATION Lines 2–4 precompute values useful during the execution of the processor and will be discussed later.

Algorithm 4.2 The sub-circuit checking Zasm code execution.

```
1: Inputs: $zasm, u, t, \mathrm{pk}_{me}, args, [r^{\mathrm{in}}], [r^{\mathrm{out}}], [R^{\mathrm{pr}}]$ as in Alg. 4.1
2: for $i \in \{0, \ldots, N_{\mathrm{fresh}} - 1\}$ do
3:     $oid_i \leftarrow H_3(i \parallel u \parallel R^{\mathrm{pr}}_{3i})$; $f_i \leftarrow H_5(i \parallel u \parallel R^{\mathrm{pr}}_{3i+2})$
4:     $\mathrm{sk}_i \leftarrow H_4(i \parallel u \parallel R^{\mathrm{pr}}_{3i+1})$; assert $\neg\mathrm{isUser}(\mathrm{derivePk}(\mathrm{sk}_i))$
5: $state_0 \leftarrow ([r^{\mathrm{in}}], (\mathrm{pk}_{me}, args, 0, \ldots, 0), [oid], [\mathrm{sk}], [f])$
6: for $i \in \{0, \ldots, N_{\mathrm{cycles}} - 1\}$ do
7:     $state_{i+1} \leftarrow \mathrm{evalInst}(zasm_i, state_i)$
8: check output state: $state_{N_{\mathrm{cycles}}} \overset{!}{=} ([r^{\mathrm{out}}], \cdot, \cdot, \cdot, \cdot)$
```

STATE The processor state $state = ([r], regs, [oid], [\mathrm{sk}], [f])$ consists of the current plain records $[r]$ of the N_{obj} involved objects (some of which may be dead), the array *regs* of N_{regs} register values, and three lists $[oid], [\mathrm{sk}], [f]$ of precomputed values. Line 5 initializes this state, where the sender address pk_{me} is (by convention) placed in the first register, followed by *args* and zero padding.

CYCLES Line 7 uses the sub-circuit $\mathrm{evalInst}(zasm_i, state_i)$ to capture how the processor executes a single instruction $zasm_i$ on input state $state_i$. Similar to [82], as proof circuits do not allow for control flow, evalInst evaluates the result of *each* possible instruction in parallel and then selects the correct result to update $state_i$ according to the instruction $zasm_i$ using a multiplexer. To reduce the proof circuit size, many parts of the computation (e.g., register and object accesses; see shortly) are shared amongst instructions.

Analogously to [82], Alg. 4.2 unrolls the processor cycles by chaining N_{cycles} copies of evalInst for a system parameter N_{cycles}, supporting any Zasm program with at most N_{cycles} instructions (programs with less than N_{cycles} instructions are padded with **NOOP** instructions). As a consequence, any function of a Zapper class may consist of at most N_{cycles} instructions. However, we note that such limits on the execution length are already commonly used in existing smart contract systems (e.g., see Ethereum's block gas limit).

FINAL STATE Finally, Line 8 checks whether the plain records in the final state $state_{N_{\mathrm{cycles}}}$ match the expected plain records $[r^{\mathrm{out}}]$.

4.6.2 *Evaluating Instructions*

We next discuss how evalInst evaluates a single processor cycle.

REGISTER ACCESS To load values from or store values to a register, we use a linear loop to select the target register based on its index. As N_{regs} is small in practice, this step is relatively efficient.

BASIC OPERATIONS Arithmetic operations work on the *regs* list and mostly correspond to the native field operations of the proof circuit. However, we impose additional checks to prevent unintended over- or underflows of field elements where necessary. For example, native addition or subtraction inside the proof circuit wraps at the field prime ($\approx 2^{255}$ in our implementation, see §4.8), which may be unexpected. We hence restrict values of type **`Uint`** to be in $[0, 2^{120})$ and reject any operation leading to a result outside this range. Values of type **`Long`** are not restricted, but cannot (by the type system) be used in arithmetic operations. [2]

LOADING AND STORING OBJECT DATA To implement **`LOAD`**, we perform a linear lookup over the plain records $[r]$ to find the targeted object id and field. We proceed analogously for **`STORE`**, **`CID`**, and **`PK`**.

Smart contracts typically only access few objects in a transaction. Therefore, the number of objects N_{obj} in $[r]$ can be set to a small constant in practice, making the above lookup relatively efficient.

As in [84], the memory of the Zapper processor is stored in a Merkle tree. However, by prefetching the memory of *few* input objects in advance (Alg. 4.1), Zapper induces significantly less overhead than checking a Merkle tree memory access in *each* processor cycle.

DESTROYING OBJECTS For the **`KILL`** instruction, the circuit simply marks the targeted record as dead by setting its *oid* component to 0 and its owner public key pk_{owner} to pk_{me}. Analogously to **`STORE`**, the targeted record is found using a linear lookup.

CREATING OBJECTS The **`NEW`** instruction creates an object by initializing a dead record with a new object id and secret key. Following the generic construction in Eq. (4.1), the object id oid_i of the i-th new object created

2 The 120 bits of a **`Uint`** value are not sufficient to provide collision-resistance of values produced by **`FRESH`**. Hence, we use a separate data type **`Long`** for these values

in *zasm* is derived as shown in Line 3. Here, u is the unique seed for the current transaction, and R is a value chosen uniformly at random and provided to ϕ as a private input. Line 3 ensures that oid_i is globally unique, even if R is selected by a malicious user (*uniqueness*). This is important, as creating a new object o whose object id matches a pre-existing object o' would allow hijacking o' and violate correctness. Further, *oid* is hidden from all other users (*secrecy*), thereby ensuring data privacy.

Analogously, the secret key sk_i of the i-th new object is derived as shown in Line 4. Here, to ensure that sk_i indeed corresponds to an object account, the Zapper client repeatedly samples uniform randomness R^{pr}_{3i+1} until the assertion in Line 4 holds.

FRESH VALUES The **FRESH** instruction computes a secret and unique value f. The i-th such unique value f_i is computed as shown in Line 4 (following Eq. (4.1)), for unique seed u and uniformly random R.

PRECOMPUTATION The computations of oid_i, sk_i, and f_i are based on cryptographic hash functions H_i, which are relatively expensive to evaluate within the proof circuit. Hence, instead of computing these in *each* processor cycle, we precompute a fixed number N_{fresh} of these values in advance (see Lines 2–4 in Alg. 4.2). Whenever such a value is required in *zasm*, we select the next unused value, assuming a Zapper function requires at most N_{fresh} such values.

DISCUSSION: UNIVERSALITY Generally, zk-SNARKs require a trusted setup, which either depends on the proof circuit (*non-universal* schemes) or not (*universal* schemes). By emulating processor execution, Zapper can use the same proof circuit to verify execution of arbitrary Zasm programs (respecting the relevant bounds such as N_{cycles}). This allows Zapper to be instantiated with an efficient non-universal SE-SNARK scheme such as GM17 [44] (see §4.8), without requiring a trusted setup per program.

An alternative design could use a universal scheme such as Marlin [98] to dynamically build a separate proof circuit per Zasm program, again avoiding a trusted setup per program. Unfortunately, Marlin is significantly more expensive than GM17: proof generation for a circuit of the same size (2 million R1CS constraints, see §4.9.3) takes 17.1 s using GM17 and 116.7 s using Marlin. While using Marlin would avoid processor emulation overhead, we expect this does not compensate for its higher cost: Zapper's proof circuit size is dominated by cryptographic operations, most of which cannot be significantly reduced (see Fig. 4.6a and §4.9.2). Still, instantiating Zapper with a universal zk-SNARK presents an interesting trade-off: Despite lower performance, such a system should allow easier extension to more complex Zasm instructions without requiring a new trusted setup for every extension.

4.7 SECURITY PROPERTIES

In this section, we discuss the privacy, correctness, integrity, and availability properties ensured by Zapper. We note that by construction, Zapper also ensures access control as defined in §4.3.2.

ATTACKER MODEL We consider an active adversary which statically corrupts a set of users and can intercept transactions by honest users. It can craft arbitrary transactions from scratch or by modifying intercepted transactions. See App. A.5 for a formal definition.

4.7.1 *Privacy*

Zapper achieves both data and identity privacy. In particular, only users with access to the secret key of an object's owner can observe when and how the object is created, read, modified, or destroyed.

IDEAL WORLD To formalize our notion of privacy, we introduce an ideal world specifying the information available to each user. We sketch this ideal world below (see App. A.5 for a formal definition).

The ideal world maintains the plaintext state of all objects and allows users to make function calls, which are executed according to the semantics of Zasm. When a user executes a function $C.f$ with arguments *args*, other users of the system only learn the following:

- Any user learns that $C.f$ is called and that the conditions of all **`REQ`** instructions are satisfied.
- All users who *can access* (see below) an object involved in a **`LOAD`**, **`STORE`**, **`KILL`**, **`NEW`**, **`CID`**, or **`PK`** operation learn the object id and the loaded (for **`LOAD`**) or stored (for **`STORE`**) value.

In particular, reading and writing fields is possible without revealing the target object's identity to any users without access to the object. Further, the arguments *args*, the return value, and the identity of the sender account are not visible, unless these are explicitly revealed to some other user by **STORE**. The same applies to any intermediate results of arithmetic operations.

ACCESSIBLE OBJECTS We say that user U *can access* object o if U *knows* the secret key of o's owner. In particular, U *knows* not only its own or shared user secret keys, but also the secret keys of objects o if (i) U *knows* the secret key of o's owner, or (ii) U created o (and thereby, its secret key) but is not necessarily its current owner. For example, in Fig. 4.2, Bob knows $\mathrm{sk}_{\mathrm{Bob}}$, $\mathrm{sk}_{\mathrm{shared}}$, and sk_{dex} and can hence access *dex* and both coins (1€ and 1$). See App. A.5 for details.

Recall that Zapper prohibits changing the owner of objects with an address (see `@has_address`) at runtime. This prevents unexpected privacy leaks: if U owns o which in turn owns o', then changing the owner of o would not invalidate U's access to o', as U would still know the secret key of o.

PRIVACY Thm. 4.1 informally states our privacy notion.

Theorem 4.1 (Privacy, informal)**.** *From transactions created by honest users, the adversary cannot learn more than in the ideal world.*

In App. A.7 (Thm. A.5), we formalize and prove Thm. 4.1. At a high level, privacy is ensured by the zero-knowledge property of the SE-SNARK, the key-privacy of the encryption scheme, and the pseudorandomness of H_i (see §4.4).

PRACTICAL CONSIDERATIONS The location or IP address from which a transaction is submitted may leak the identity of the sender in practice. Hence, as in similar systems [25], users may want to submit transactions via an anonymous overlay network such as Tor [116].

4.7.2 *Correctness, Integrity, Availability*

We now informally present the remaining security properties (as introduced in §4.2), which we formalize in App. A.5–A.6.

Theorem 4.2 (Correctness, informal)**.** *The adversary cannot violate the logic specified in contracts registered at the Zapper executor.*

Theorem 4.3 (Integrity, informal)**.** *Valid transactions cannot be modified "in flight" or replayed by the adversary.*

Theorem 4.4 (Availability, informal). *Honest users can realize valid transactions unless the adversary actively interferes.*

At a high level, correctness is enforced by the construction of ϕ and the soundness property of the SE-SNARK. Further, the non-malleability of SE-SNARKs prevents transactions from being tampered with, and checking the uniqueness of serial numbers prevents replay attacks. Note that since our attacker model allows the adversary to intercept every transaction, it can always block the *current* transaction. However, if the adversary does not interfere with a transaction, Thm. 4.4 states that it will always be accepted. In particular, the adversary cannot "lock" an object owned by an honest user by publishing a colliding serial number or refusing to share the decryption key. This prevents the "Faerie Gold" attack of Zerocash (allowing attackers to permanently block coins of honest owners) [26] and two attacks on ZEXE (discussed in §4.11).

4.8 IMPLEMENTATION

We implemented Zapper in an end-to-end system consisting of a Python frontend (3k LoC) exposed to application developers, and a Rust backend (4k LoC) performing cryptographic tasks and relying on the efficient arkworks libraries [117]. Our implementation includes an idealized centralized ledger, which can be replaced by a shared ledger in an actual deployment. Below, we describe how the cryptographic primitives are instantiated in our implementation. All these primitives have been used in existing systems.

PROVING SCHEME AND ELLIPTIC CURVES We use the simulation-extractable GM17 [44] zk-SNARKs over the pairing-friendly Barreto-Lynn-Scott [118] curve BLS12-381 introduced in Zcash [26]. The constraints of the proof circuit ϕ are then expressed in the scalar field $\mathbb{F}_q$ of BLS12-381, where $q \approx 2^{255}$. To allow for efficiently emulating the Zapper processor in ϕ, Zasm code operates on values in $\mathbb{F}_q$.

Like Zcash, we rely on curve embedding to efficiently evaluate cryptographic primitives (see below) within ϕ. In particular, our primitives use the Jubjub [119] twisted Edwards curve, whose base field matches the scalar field $\mathbb{F}_q$ of BLS12-381. This allows us to natively evaluate Jubjub curve operations in ϕ.

HASH FUNCTIONS Like ZEXE [39], we rely on Pedersen and Blake2s hashes. In particular, we instantiate the hash function H for the object tree

Table 4.2: Evaluated example Zapper applications and classes. We indicate the number of functions (#fun), and the min/max number of Zasm instructions in the functions of each class (inst).

App	Description	Classes	#fun (inst)
Auction	A private decentralized coin auction.	Auction$^{\$}$	3 (33–48)
Coin	A private untraceable coin.	Coin (Fig. 4.1)	5 (6–29)
Exchange	A private decentralized coin exchange.	DexOffer$^{\$}$ (Fig. 4.1)	3 (18–36)
Heritage	A heritable coin wallet with anonymous heirs and private shares.	Share Wallet$^{\$}$	2 (7–8) 7 (9–80)
Reviews	A double-blind peer-review system for academic papers.	Review Result Paper	3 (8–25) 3 (10–11) 4 (17–32)
Tickets	A public transport ticketing system with untraceable multi-journey tickets.	TicketProof Ticket$^{\$}$	1 (7) 4 (10–25)
WorkLog	A system for aggregate working hours reports hiding check-in/-out times.	Aggregated WorkLog	1 (7) 4 (9–23)

$^{\$}$ makes use of the Coin class

by the Pedersen hash [26, §5.4.1.7] with 4-bit windows over Jubjub. H is collision-resistant assuming it is hard to compute discrete logarithms in Jubjub [120]. As Pedersen hashes do not provide pseudo-randomness, we use the pseudo-random Blake2s hash function [121] to instantiate $H_i(x) :=$ Blake2s$(i \parallel x)$ as per §4.4.

encryption Like the Dusk Network [122], we use a hybrid encryption scheme based on Poseidon [123] and ElGamal [124]. In particular, to encrypt a plain record r, we first select a random curve point k on Jubjub and encrypt r with key k using Poseidon [123] in the DuplexSponge framework [125, 126]. Then, we encrypt k using ElGamal [124] over Jubjub with the owner's public key. As ElGamal encryption is key-private [115], this hybrid scheme is also key-private.

4.9 EVALUATION

We now evaluate our implementation of Zapper (§4.8), demonstrating that it is highly efficient. All our experiments are conducted on a desktop machine with 32 GB RAM and 12 CPU threads at 3.70 GHz.

4.9.1 *Example Applications*

To demonstrate the expressiveness of Zapper, we implemented the 7 applications described in Tab. 4.2 in Zapper's Python frontend using a total of 12 classes. The applications span a variety of domains and correspond to realistic use cases. The Coin and Exchange apps (see Fig. 4.1) closely follow the "user-defined asset" and "intent-based DEX" examples of ZEXE [39]. In contrast to ZEXE, where these apps are implemented as low-level predicates, they are naturally expressed in Zapper's frontend. The other apps are our creations. Being a core component, the Coin class is used across multiple apps.

In all applications, Zapper's data and identity privacy properties are key. For example, "Ticket" allows travelers to punch multi-journey tickets valid for a specific duration after punching, while preventing ticket holders to be traced across journeys.

4.9.2 *Performance*

We now evaluate the performance of Zapper.

PARAMETERS For the following experiment, we instantiate the parameters of Zapper as shown in Tab. 4.3a. Here, N_{fields} is the maximum number of fields per object, excluding `owner`. The Merkle tree height N_{height} is sufficiently large for a real-world deployment and matches the height in Zcash [26]. The other parameters were chosen to support the apps in Tab. 4.2 with a comfortable margin.

The choice of parameters is subject to a trade-off: larger values enable more complex applications but induce more overhead. In practice, Zapper can support multiple combinations of parameters and prepare a separate proof circuit for each combination, enabling Zapper to select the smallest parameters sufficient to enable any given application. However, it is not obvious how to allow setting N_{height} dynamically, as all applications must operate on the same Merkle tree. To help predict the performance of future applications, §4.9.3 discusses the effect of individual parameters on performance.

SCENARIOS For each application in Tab. 4.2, we create a *scenario* consisting of multiple transactions interacting with the application. For example,

Table 4.3: Evaluation parameters and runtimes.

(a) Values for parameters.

Tree height	N_{height}	32
Objects in tx	N_{obj}	4
Fresh values	N_{fresh}	4
Processor cycles	N_{cycles}	100
Registers	N_{regs}	10
Object fields	N_{fields}	9

(b) Runtimes for example scenarios.

	Step	**Time (std. dev.)**	
one-time	setup	37.207 s	
per app	compile	0.007 s	($\pm$0.003 s)
per tx	create	21.639 s	($\pm$0.152 s)
	verify	0.027 s	($\pm$0.003 s)

for Exchange we first mint two coins and then run `create` and `accept` transactions as visualized in Fig. 4.2.

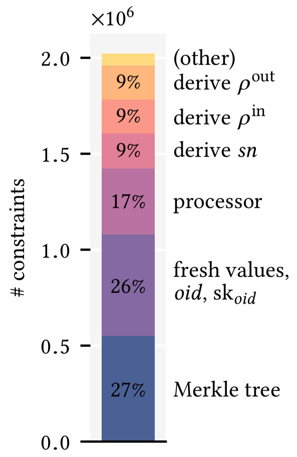
RUNTIMES Tab. 4.3b summarizes the performance of Zapper for our scenarios, showing that Zapper transactions are very efficient.

Before executing any transactions, we first initialize a new Zapper ledger ("setup" in Tab. 4.3b). This global one-time step includes a trusted setup phase for the GM17 [44] zk-SNARKs, which is relatively expensive and dominates the runtime of the step (99.87%).

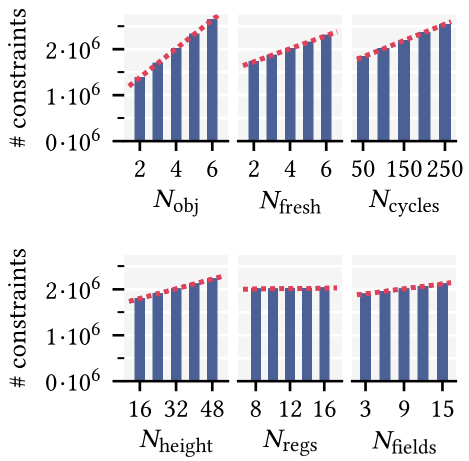
Next, we compile all applications in Tab. 4.2 to Zasm code and register this code at the Zapper assembly storage. As shown in Tab. 4.3b ("compile"), compiling and registering a single application is very efficient as it does not involve any cryptographic operations.

Finally, we execute the scenarios on the Zapper ledger. Executing a transaction consists of two steps. First, the Zapper client locally creates the transaction (§4.5.2, "create"). The runtime of this step is dominated by zk-SNARK generation (99.97% on avg.). Second, the Zapper executor processes this transaction (§4.5.3, "verify"), which is dominated by zk-SNARK verification (59.1%) and Merkle tree updates (40.6%). While the first step is more expensive, it is only executed by the client. In contrast, the second step is significantly cheaper. This is important, as it will be replicated across many machines when deploying Zapper to a shared ledger.

RUNTIME DISCUSSION As shown by the low standard deviations in Tab. 4.3b, the runtimes for creating and verifying transactions are very consistent across all scenarios: because the proof circuit for the zk-SNARK in a transaction is independent of the application, the dominating proof generation times are nearly identical.



(A) Breakdown of component sizes for the parameters in Tab. 4.3a.



(B) Circuit size when varying individual parameters (■) and estimate (┅) computed by the empirical fit in Eq. (4.5).

Figure 4.6: Number of R1CS constraints in the proof circuit ϕ (Alg. 4.1).

Our transaction runtimes are in line with ZEXE [39]. The authors report that transactions require 52.5 s to generate and 0.046 s to verify on a similar machine (3.00 GHz, 24 threads). While we used $N_{\text{obj}} = 4$ and the logic of contracts in Tab. 4.2, the ZEXE evaluation assumed 2 inputs and 2 outputs, and empty predicates.

TRANSACTION SIZE Zapper transactions are small, consisting of only 3 312 bytes regardless of the called function. This is comparable to ZEXE, whose transactions consist of 968 bytes.

4.9.3 *Proof Circuit Size*

The runtimes for zk-SNARK setup and proof generation, which are the dominating parts of "setup" and "create" in Tab. 4.3b, are linear in the size of the proof circuit. We next analyze this size, measured in the number of R1CS constraints.

SIZE OF INDIVIDUAL COMPONENTS For the parameters in Tab. 4.3a, the proof circuit (Alg. 4.1) consists of $2.02 \cdot 10^6$ constraints (regardless of the application). In Fig. 4.6a, we show the sizes of the individual components. The largest two components are the checks of Merkle tree certificates (Line 13 in Alg. 4.1) and preparation of precomputed values (Lines 2–4 in Alg. 4.2), as they involve the evaluation of expensive cryptographic hash functions. This is followed by the emulation of processor cycles (Lines 5–7 in Alg. 4.2), the derivations of serial numbers sn, nonces ρ^{in}, and nonces ρ^{out} (Alg. 4.1).

EFFECT OF PARAMETERS We next measure the size of the proof circuit for different parameters. This allows gauging the performance of Zapper when parameters are selected dynamically from a set of prepared parameters. In each of the subplots in Fig. 4.6b, we show the number of constraints when varying a single parameter while setting all other parameters to the values in Tab. 4.3a. Fig. 4.6b indicates that N_{obj} has the biggest impact, while N_{regs} and N_{fields} have negligible effects. Overall, the main proof circuit has asymptotic size:

$$\mathcal{O}\Big(\underbrace{N_{\text{obj}}(N_{\text{height}} + N_{\text{fields}})}_{\text{Lines 8–16 \& 18–20 in Alg. 4.1}} + \underbrace{N_{\text{fresh}}}_{\substack{\text{Lines 2–4}\\ \text{in Alg. 4.2}}} + \underbrace{N_{\text{cycles}}(N_{\text{obj}} N_{\text{fields}} + N_{\text{regs}} + N_{\text{fresh}})}_{\text{Lines 5–8 in Alg. 4.2}} \Big).$$

To predict the performance of Zapper for given parameters, we have further estimated the constants hidden in the above formula using an empirical least-square fit. We find that the number of R1CS constraints in the proof circuit can be predicted as:

$$\begin{aligned}&3\,400 + N_{\text{obj}}\left(160\,000 + 3\,300\,N_{\text{height}} + 1\,900\,N_{\text{fields}}\right) + 130\,000\,N_{\text{fresh}} \\ &+ N_{\text{cycles}}\left(1\,600 + 26\,N_{\text{obj}}\,N_{\text{fields}} + 24\,N_{\text{regs}} + 120\,N_{\text{fresh}} + 76\,N_{\text{obj}}\right).\end{aligned} \tag{4.5}$$

As indicated in Fig. 4.6b, this prediction is very accurate.

4.10 LIMITATIONS

We now discuss limitations of Zapper.

First, Zapper only supports Zasm programs respecting its parameters N_{cycles}, N_{obj}, N_{fresh}, N_{fields}, and N_{regs}. If these parameters limit expressivity, developers can increase them (see §4.9.2).

As discussed in §4.3.1, Zapper does not natively support control flow, but if-then-else branches can always be rewritten as conditional assignments, and bounded loops can be unrolled. While unbounded loops are not supported, these are already discouraged in non-private smart contracts [91] and can instead be split into individual, bounded-length transactions. Further, Zapper disallows pointer arithmetic and self-modifying code, which are however uncommon in smart contracts. Also, Zasm programs with recursion are disallowed and must be restructured. We expect this is often feasible, e.g. by rewriting tail recursion into loops or by splitting recursive calls into individual non-recursive transactions.

A fundamental limitation of Zapper is that it only allows users to create transactions for which the data of all accessed objects is known. In particular, contracts cannot privately communicate "amongst each other" while keeping the communicated data hidden from all users. To enable this, Zapper would need to leverage additional cryptographic primitives such as homomorphic encryption.

Finally, some applications such as private machine learning are not suitable for Zapper, but can be realized using SMC or FHE. However, these techniques are generally less scalable in terms of the number of involved parties, communication, or computation.

4.11 COMPARISON TO ZEXE

We now elaborate on the shortcomings of ZEXE [39] compared to Zapper (see also §4.1). ZEXE specifies smart contracts using records and predicates. It provides strong data, identity, and function privacy based on nested zk-SNARKs and ideas from Zerocash [25].

APPLICATION VULNERABILITIES We have discovered two vulnerabilities in ZEXE's motivating example of a DEX [39, §I-A], which allow an attacker to lock a coin belonging to another user. Both attacks have been confirmed by the authors of ZEXE ([127, Acknowledgments] and private correspondence). Note that our notion of availability (§4.7) prevents such attacks by design.

First, as ZEXE only stores commitments of data, transferring data to another user requires out-of-band communication. As this can be denied by a malicious user, DEX is subject to a "denial-of-funds" attack, where the attacker accepts an offer but refuses to share the information required to receive the attacker's coin. To prevent this attack, the ZEXE authors recommended adapting DEX to store the encrypted output record in a public memorandum field and extending the predicates to check for correct encryption [127, Remark 6.1]. However, only developers intimately familiar with key-private and NIZK-friendly encryption can implement this securely and efficiently. Even if cryptographic experts provide according cryptographic primitives, developers still need to decide whether to use these, depending on the application (note that the attack does not exist for the coin itself, but only when the coin is used in a DEX). In contrast, Zapper by design uses an appropriate encryption scheme instead of commitments.

Second, we have identified a "lock-out" attack on the DEX application. Like Zapper, ZEXE pads input records by dead records. However, unlike Zapper, ZEXE does not enforce that their serial nonces ρ^{in} are globally unique (see Line 15 in Alg. 4.1). Thus, an attacker knowing the shared address secret key of a DEX record can block access to the record by consuming a dead record with a conflicting serial nonce and serial number, thus blocking the coin to be traded by the DEX indefinitely. We expect that our attack can be prevented in ZEXE by constraining serial nonces of dead records.

LACK OF MODULARITY ZEXE obstructs modular development, as co-operating applications must typically be mutually aware of each other to ensure that their logic cannot be violated in the future.

For example, ZEXE's motivating example of a DEX [39, §I-A] introduces a tight coupling between DexOffer and Coin. Specifically, to prevent adversaries from creating coins out of thin air, their *birth predicate* (which must be satisfied when creating a new coin) ensures that coins can only be created in exchange for existing coins (identified by their birth predicate). However, because a DexOffer record cannot "own" a coin record (ZEXE has no concept of ownership), a newly created DexOffer consumes the coin to be traded. In turn, accepting a DexOffer hence re-creates the previously consumed coin, which requires the DexOffer to have the same birth predicate as coin (see above), thus essentially merging both applications. Note that adapting the Coin birth predicate to allow consuming non-merged DexOffer objects would require trusting that DexOffer does not create coins out of thin air.

Following the above pattern, all potential applications using coins must be anticipated and implemented in advance—a severe limitation in practice.

In contrast, Zapper's object ownership feature and access control policies allow classes to be developed independently and modularly.

TRUSTED SETUP AND USABILITY ZEXE requires a separate trusted setup for *each* application, which when performed by dishonest parties allows violating correctness. [3] In contrast, Zapper only requires a single trusted setup for its application-agnostic proof circuit ϕ. Also, ZEXE relies on a non-standard programming model in terms of *predicates*, while the programming model of Zapper is closer to the most widely used smart contract language Ethereum.

FUNCTION PRIVACY Unlike Zapper, ZEXE hides the function being executed in a transaction. However, this is often not required in practice (see Tab. 4.2). Still, future work could extend Zapper to function privacy by providing Zasm instructions as private inputs to the proof circuit and performing class registrations in zero-knowledge.

3 This problem can be partially mitigated by an expensive SMC, assuming trustworthy participants can be found for every new application.

4.12 RELATED WORK

We next discuss works related to Zapper.

PRIVATE CRYPTOCURRENCIES We have already discussed anonymous payment systems in §1.1. In contrast to Zapper, these systems focus on payments only and do not support general smart contracts. However, Zapper does rely on the techniques of Zerocash [25, 26] to hide the objects accessed in a transaction.

PRIVATE SMART CONTRACTS Various works bring privacy to smart contracts. Hawk [32], Arbitrum [33], Ekiden [34], and FastKitten [35] assume a strong trust model by relying on trusted managers or hardware. In contrast, Zapper only relies on a single trusted zk-SNARK setup. While zkHawk [36] and V-zkHawk [37] weaken the trust assumption of Hawk, they require interactive parties. The zkay and ZeeStar systems introduced in Chapters 2–3 provide data privacy for smart contracts with weak trust assumptions, but do not target identity privacy and leak the accessed memory locations. Similarly, SmartFHE [38], does not provide identity privacy. We have already discussed ZEXE [39] separately in §4.11.

ZERO-KNOWLEDGE ROLLUPS Complementary to Zapper, *ZK rollups* such as StarkNet [128], zkSync [129], and Aztec [130] combine multiple smart contract transactions into a single one using NIZK proofs. However, to date, StarkNet and zkSync do not provide privacy [131, 132]. While the announced Aztec "ZK-ZK-Rollup" system [133] aims to achieve *private* rollups, it has not yet been released at the time of this writing.

ZERO-KNOWLEDGE PROCESSORS The idea of executing a processor in zero-knowledge has been thoroughly studied before Zapper. For instance, BubbleRAM [134], BubbleCache [135], and ZKarray [136] present zero-knowledge processors with efficient RAM. However, unlike Zapper, they target an interactive setting.

Similar to Zapper, the TinyRAM line of work [82, 83, 84] emulates a processor inside (non-interactive) zk-SNARKs. As discussed in §4.6, the Zapper processor applies techniques introduced by these works. Zapper could potentially be extended, e.g., to target a von Neumann architecture, allowing powerful techniques such as self-modifying code [83]. However, as we demonstrate in §4.9, the Zapper processor already supports realistic applications.

4.13 SUMMARY

In this chapter, we presented Zapper, a smart contract system providing both data and identity privacy. Zapper allows developers to express smart contracts in an intuitive frontend and executes these on a custom distributed ledger. It supports important applications such as anonymous coins and private decentralized exchanges.

Zapper is highly efficient and achieves data and identity privacy, correctness, access control, integrity, and availability.

5 CONCLUSION AND FUTURE WORK

In this thesis, we explored how to ensure privacy for smart contracts on permissionless blockchains. In contrast to previous work, we have followed a programming language approach to design and implement systems for private smart contracts. The three presented systems provide different levels of privacy and expressivity.

In Chapter 2, we presented the zkay system, which relies on asymmetric encryption and NIZK proofs to hide the data involved in transactions. We introduced the zkay programming language, whose privacy types allow specifying and tracking owners of private data, as well as preventing implicit information leaks. Our automatic compilation of zkay to Ethereum contracts enables the deployment of practical privacy-preserving applications.

Being the first work of its kind, zkay does not allow computation on unknown data. In Chapter 3, we presented the ZeeStar system, which extends zkay by homomorphic encryption in order to relieve this fundamental restriction. We discussed how to extend the zkay privacy type system to support computation on unknown data, and how to automatically combine zk-SNARKs with homomorphic encryption in an efficient manner. Unlike zkay, ZeeStar allows developers to readily express key applications such as private coins.

Chapters 2–3 focused on data privacy. In Chapter 4, we investigated how to achieve an even stronger notion of privacy, where also the accessed memory locations and parties involved in a transaction are hidden. In particular, we presented the Zapper system, which conceptually is an extension of Zerocash [25] allowing "coins" to be programmed with custom logic. Zapper's custom assembly language and access control mechanism support modular development of applications while preventing malicious code from arbitrarily interfering with other applications. We realize Zapper by combining modern zk-SNARKs with NIZK-friendly encryption.

We implemented all three systems and demonstrated their efficiency on several example applications. By making the implementations publicly available under permissive licenses, we facilitate extensions of our work by other researchers and practitioners.

Throughout this thesis, we employed core techniques from the area of programming languages, including type systems, program analysis, and compilation. We used these techniques to automatically combine advanced cryptographic primitives, allowing us to make their instantiation transparent to application developers. We believe such an approach is crucial in order to allow developers without cryptographic expertise to implement privacy-preserving smart contracts. We hope that the research community continues to make smart contract privacy more accessible.

FUTURE WORK While this thesis makes several important contributions, we have by far not explored the posed research question to completeness. Below, we suggest several items for future work.

Coalescing all Systems: The presented systems have incomparable properties. While the privacy type system of zkay and ZeeStar prevents implicit information leaks, there is no such mechanism in Zapper. On the other hand, Zapper provides identity privacy, while zkay and ZeeStar only provide data privacy. Further, ZeeStar supports computation on unknown data, which is not possible in Zapper. Combining all these features in a single system providing identity privacy, checking for implicit leaks, and allowing for computation on unknown data is an interesting item for future work.

Extended Privacy Types: The privacy type systems of zkay and ZeeStar are limited as each value can only be owned by a single address. By extending the privacy types to support multiple owners, we can likely support more interesting applications. For instance, future work could investigate how to support group annotations of the form `@{Alice, Bob}`, indicating that the respective value can be accessed by both Alice and Bob.

Ownership Transfer: The address of an owner field cannot be changed in zkay or ZeeStar once the contract has been constructed. This is in contrast to Zapper, which allows dynamically changing the owner of an object. Unlike Zapper, zkay and ZeeStar feature a fine-grained model of privacy, where different fields of the same contract and different mapping entries of the same mapping can be owned by different addresses. Hence, supporting ownership transfer in zkay or ZeeStar requires statically determining all values owned by an owner field. As we already discuss in §2.9, such an extension is left to future work.

Reducing Trust: Being based on non-universal NIZK proof schemes, our implementations of the zkay and ZeeStar systems require a separate trusted setup for each contract. Recently, several schemes with universal [95, 96, 97, 98] or transparent setup [99] have been proposed. Instantiating such schemes in zkay and ZeeStar would relieve the requirement of a per-contract setup. However, this is challenging, as the resulting system should still allow for acceptable Ethereum gas costs, and some schemes [97] are not targeted at the R1CS abstraction used in this work.

Extending the Zasm Processor: While supporting a variety of applications, the instruction set of the Zasm processor is deliberately kept at a minimum complexity. As a result, Zapper currently does not support control flow, pointer arithmetic, or self-modifying code. In future work, we propose to extend the Zasm processor to support these and other features. To this end, one can likely re-use ideas from existing work on NIZK processors [82, 83, 84].

Improving Scalability: The object tree of Zapper stores an ever-growing list of data, because all previous states of any object are kept indefinitely. In fact, the privacy notion of Zapper prevents the automatic deletion of outdated object data, because it is unknown which records correspond to current states. This is a problem, as a long-running ledger will eventually require an untractable amount of storage to process new transactions. In future work, we could investigate how to reduce the amount of storage, for instance by leveraging nested NIZK proofs.

A APPENDIX

A.1 SECURITY DEFINITIONS

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

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

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

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

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

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

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

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

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

Definition A.4 (zk-SNARG [45]). *A* zero-knowledge succinct non-interactive argument system *is a tuple $(Setup, Prove, Verify)$ of PPT algorithms, where $Setup(\phi)$ returns a CRS Σ and trapdoor τ, $Prove(\Sigma, \phi, x, w)$ returns a proof π for the circuit $\phi(x; w)$, $Verify(\Sigma, \phi, x, \pi)$ returns 1 iff π is considered valid, and:*

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

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

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

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

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

A.2 FORMAL SEMANTICS OF ZKAY

TYPES For each data type τ, we define the set $[\![\tau]\!]$ of values a variable of type τ may assume. For example, $[\![$**uint**$]\!] = [0, 2^{32} - 1]$, $[\![$**bool**$]\!] = \{$**true**, **false**$\}$, and $[\![$**address**$]\!] = \{0, 1\}^*$. Further, $[\![$**bin**$]\!]$ consists of symbolic representations of keys, ciphertexts, proofs and encryption randomness. Symbolic public and secret keys are of the form pk_a and sk_a, for an address a. The semantics of `mapping(`τ_1 `=>` τ_2`)` is given by a partial function from $[\![\tau_1]\!]$ to $[\![\tau_2]\!]$. For example, a value of type `mapping(uint => uint)` is $\{1 \mapsto 4, 2 \mapsto 4\}$. Reading a mapping for an uninitialized key yields undefined behavior.

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 $\Gamma = \{x$: `mapping(uint => uint@all)`, y: `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.

$$\langle \texttt{me}, \sigma \rangle \xmapsto{\sigma(\texttt{me})@\texttt{all}} \sigma(\texttt{me}) \qquad \frac{\langle e, \sigma \rangle \overset{t_1}{\mapsto} v \quad \langle \alpha, \sigma \rangle \overset{t_2}{\mapsto} a}{\langle \texttt{reveal}(e, \alpha), \sigma \rangle \xmapsto{t_1, t_2, v@a} v}$$

$$\langle \texttt{all}, \sigma \rangle \xmapsto{\text{all}@\texttt{all}} \texttt{all} \qquad \frac{\langle L, \sigma \rangle \overset{t_1}{\Vert\!\!\mapsto} l \quad \langle \alpha, \sigma \rangle \overset{t_2}{\mapsto} a}{\langle L, \sigma \rangle \xmapsto{t_1, t_2, \sigma(l)@a} \sigma(l)} \qquad \frac{\langle L, \sigma \rangle \overset{t_1}{\Vert\!\!\mapsto} l \quad \langle e, \sigma \rangle \overset{t_2}{\mapsto} v}{\langle L[e], \sigma \rangle \Vert\!\!\xmapsto{t_1, t_2, l[v]@\texttt{all}} l[v]}$$

$$\langle c, \sigma \rangle \xmapsto{c@\texttt{all}} c$$

FIGURE A.2: Semantics for selected expressions and locations. For location reads, we assume that L is typed according to Fig. 2.4.

LANGUAGE CONSTRUCTS In Fig. A.1, we summarize the 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 $(v_1, \dots, v_n)$. 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	$\langle L, \sigma \rangle \overset{t}{\Vert\!\!\mapsto} l$
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 A.1: 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 $\langle 1/0, \sigma \rangle \xmapsto{1,0,\texttt{fail}} \texttt{fail}$. 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 $\langle \text{id}, \sigma \rangle \Vert\!\!\xmapsto{\text{id}@\texttt{all}} \text{id}$. Further, Fig. A.2 provides semantics for mapping lookups.

Expressions: For simplicity, our expressions do not support side effects. In general, the trace t when evaluating expression e contains (as its last entry)

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

FIGURE A.3: Semantics for contract functions.

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

$$\frac{\begin{array}{cc} \langle p, \sigma\rangle \xmapsto{t_0} \mathrm{Proof}_\phi(R; v_{1:n}; v'_{1:m}) & \begin{array}{c}\langle e_1, \sigma\rangle \xmapsto{t_1} v_1 \\ \dots \\ \langle e_n, \sigma\rangle \xmapsto{t_n} v_n\end{array} \\ \phi(v_1, \dots, v'_m) = 1 & \end{array}}{\langle \mathtt{verify}_\phi(p, e_1, \dots e_n), \sigma\rangle \xmapsto{t_0, t_1, \dots, t_n, 1@\mathtt{all}} \sigma}$$

FIGURE A.4: Semantics for selected statements. Typed by Fig. 2.5, where α is the privacy type of L.

the value v resulting from evaluating e, with privacy 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. A.2 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 $g(e_1, \dots, e_n)$ is standard and thus omitted. The trace of the evaluation only consists of the return value and the traces of evaluating the arguments.

Fig. A.3 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. A.4 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

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

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

FIGURE A.5: Semantics for transactions. Here, the privacy level α_i is **all** if the i-th argument is public, and a otherwise.

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. A.5 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**.

A.3 PRIVACY OF ZEESTAR

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

To formalize the information the attacker is expected to learn, we again use the observable ideal-world traces defined for zkay (§2.6.2). In the

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

```
1: Run ZeeStar to transform $C$ to $\bar{C}$
2: For every proof circuit $\phi$ in $\bar{C}$: $(\Sigma_\phi, \tau_\phi) \leftarrow \mathrm{Setup}(\phi)$
3: For every account $\alpha$, generate a fresh key pair $(pk_\alpha, sk_\alpha)$
4: Collect all public keys in the mapping $\mathrm{pk}(\alpha) \mapsto pk_\alpha$
5: Create the initial empty state $\bar{\sigma}_0$ for $\bar{C}$ and $\sigma_0$ for $C$
6: $\mathcal{P}.\mathrm{Init}(C, \mathrm{pk}, \{sk_\alpha\}_{\alpha \notin \mathcal{A}}, \{\Sigma_\phi\})$   $\mathcal{S}.\mathrm{Init}(C, \mathrm{pk}, \mathcal{A}, \{\Sigma_\phi\}, \{\tau_\phi\})$
7: $\mathcal{E}.\mathrm{Init}(C, \mathrm{pk}, \{sk_\alpha\}_{\alpha \in \mathcal{A}}, \{\Sigma_\phi\})$
8: for $i = 1, \dots, n$ do:
9:    if $\mathrm{sender}[tx_i] \in \mathcal{A}$ then
10:       $\bar{tx}_i \leftarrow \mathcal{E}.\mathrm{Tx}(C, \bar{\sigma}_{i-1})$
11:                                  $tx_i \leftarrow \mathrm{GetIdeal}(\bar{tx}_i, \{\Sigma_\phi\}, \bar{C}, \bar{\sigma}_{i-1}, \{sk_\alpha\}_{\alpha \in \mathcal{A}})$
12:   else
13:       $\bar{tx}_i \leftarrow \mathcal{P}.\mathrm{Tx}(C, \bar{\sigma}_{i-1}, tx_i)$   $t \leftarrow \mathrm{obs}_{\mathcal{A}}(C, \sigma_{i-1}, tx_i)$
                                                  $\bar{tx}_i \leftarrow \mathcal{S}.\mathrm{Tx}(C, \bar{\sigma}_{i-1}, t)$
14:   Run $\bar{tx}_i$ on $\bar{C}$, $\bar{\sigma}_{i-1}$ to get $\bar{\sigma}_i$ and $tx_i$ on $C, \sigma_{i-1}$ to get $\sigma_i$
15:   $\mathcal{E}.\mathrm{Notify}(\bar{tx}_i)$
16: return $\mathcal{E}.\mathrm{Decide}()$
```

following, let $\mathrm{obs}_{\mathcal{A}}(C, \sigma, tx)$ denote the ideal-world trace observable by the parties in $\mathcal{A}$ when a transaction tx is executed on contract C in the ideal-world state σ.

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

The PPT protocol $\mathcal{P}$ captures the behavior of honest accounts. $\mathcal{P}$.Init (Line 6) registers global configuration and keys, and $\mathcal{P}.\mathrm{Tx}(C, \bar{\sigma}_{i-1}, tx_i)$ (Line 13) transforms the transaction tx_i by calling the function T_{TX} defined in Alg. 3.2 using the information previously received via $\mathcal{P}$.Init.

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

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

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

PRIVACY If $\mathcal{S}$ can be instantiated such that any adversary $\mathcal{E}$ cannot distinguish whether it is interacting with real honest parties (in $\mathrm{Real}_{\mathcal{A}}^{\mathcal{E}}$) or with simulated parties (in $\mathrm{Sim}_{\mathcal{A}}^{\mathcal{E},\mathcal{S}}$), then the system respects privacy. This is formalized in Thm. A.1.

Theorem A.1 (Privacy of ZeeStar). *Assume ZeeStar is instantiated with a randomizable (Def. A.3) and IND-CPA (Def. A.2) encryption scheme, and a zk-SNARG (Def. A.4). Let C be a well-typed ZeeStar contract and $\mathcal{A}$ any set of parties. Further, let $tx_{1:n}$ be any sequence of n transactions, where n is polynomial in the security parameter. There exists a PPT protocol $\mathcal{S}^\star$ such that for any PPT adversaries $\mathcal{E}, \mathcal{E}'$, the following advantage (Def. A.1) is negligible:*

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

Proof. Let C, $\mathcal{A}$, and $tx_{1:n}$ as in Thm. A.1. We next construct PPT simulators $\mathcal{S}_i$ for $i \in \{0, \dots, 8\}$, following the ideas of the symbolic proof in §2.6. By defining $\mathcal{S}^\star := \mathcal{S}_8$, Thm. A.1 follows from Lems. A.1–A.9 below and the triangle inequality. □

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

Lemma A.1. *For any PPT adversaries $\mathcal{E}, \mathcal{E}'$, it is*

$$Adv^{\mathcal{E}'}(Real^{\mathcal{E}}_{\mathcal{A}}(C, tx_{1:n}),\ Sim+^{\mathcal{E},\mathcal{S}_0}_{\mathcal{A}}(C, tx_{1:n})) = 0.$$

Proof. Straightforward, by construction. □

THE SIMULATOR $\mathcal{S}_1$ $\mathcal{S}_1$ is the same as $\mathcal{S}_0$, but with the following modification. Instead of executing Lines 2–3 in Alg. 3.2 (as part of $\mathcal{P}$.Tx), $\mathcal{S}_1$ reads the function, sender address, and values of the public arguments from the observable trace t. This is possible as these items are public and therefore available in t.

Lemma A.2. *For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ it is:*

$$Adv^{\mathcal{E}'}(Sim+^{\mathcal{E},\mathcal{S}_0}_{\mathcal{A}}(C, tx_{1:n}), Sim+^{\mathcal{E},\mathcal{S}_1}_{\mathcal{A}}(C, tx_{1:n})) = 0.$$

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

THE SIMULATOR $\mathcal{S}_2$ $\mathcal{S}_2$ is the same as $\mathcal{S}_1$, but we change the behavior of Line 7 in Alg. 3.2 as follows.

Whenever $\mathcal{S}_1$ creates an encryption $Enc_\alpha(T_{\text{plain}}(e))$ for a *dishonest* party $\alpha \in \mathcal{A}$ due to the first case in rule (3.18), $\mathcal{S}_2$ does not call T_{plain}, but reads the plaintext value v of e from the ideal-world trace t. As e is public, v occurs in t.

Similarly, whenever $\mathcal{S}_1$ creates an encryption $Enc_\alpha(T_{\text{plain}}(e))$ for $\alpha \in \mathcal{A}$ due to Eq. (3.19), $\mathcal{S}_2$ reads the plaintext value v of e from the ideal-world trace t. As e is revealed to $\alpha \in \mathcal{A}$, v is visible in t.

Also, whenever $\mathcal{S}_1$ calls T_{plain} due to the first case in Fig. 3.6, $\mathcal{S}_2$ reads the plaintext value v of e from the ideal-world trace t. As e is revealed to the public, v occurs in t.

Finally, when processing the rule (3.21), $\mathcal{S}_2$ reads the plaintext value v of e_1 from t. Again, as e_1 is public, v occurs in t.

Lemma A.3. *For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ the following advantage is negligible:*

$$Adv^{\mathcal{E}'}(Sim+^{\mathcal{E},\mathcal{S}_1}_{\mathcal{A}}(C, tx_{1:n}), Sim+^{\mathcal{E},\mathcal{S}_2}_{\mathcal{A}}(C, tx_{1:n})).$$

Proof. By construction, the simulators $\mathcal{S}_1$ and $\mathcal{S}_2$ output the same distribution if the values v accessed as described above are correct. By Thm. 3.1 (correctness) and because n is polynomial, this is the case with overwhelming probability. □

THE SIMULATOR $\mathcal{S}_3$ $\mathcal{S}_3$ is the same as $\mathcal{S}_2$, but we modify Eq. (3.22) as follows. If $\alpha \in \mathcal{A}$, then the value v_1 of e_1 is revealed to the adversary (due to the `reveal` expression) and hence available in the trace t. In this case, instead of calling T_{plain} in $\mathcal{S}_2$, $\mathcal{S}_3$ reads v_1 from t.

Lemma A.4. *For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ the following advantage is negligible:*

$$Adv^{\mathcal{E}'}(Sim{+}^{\mathcal{E},\mathcal{S}_2}_{\mathcal{A}}(C, tx_{1:n}), Sim{+}^{\mathcal{E},\mathcal{S}_3}_{\mathcal{A}}(C, tx_{1:n})).$$

Proof. By construction, the simulators $\mathcal{S}_2$ and $\mathcal{S}_3$ output the same distribution if the values v_1 accessed as described above are correct. By Thm. 3.1 (correctness) and because n is polynomial, this is the case with overwhelming probability. □

THE SIMULATOR $\mathcal{S}_4$ $\mathcal{S}_4$ is the same as $\mathcal{S}_3$, but instead of generating real proofs in Line 9 of Alg. 3.2, $\mathcal{S}_4$ uses SimProof (Def. A.4) to generate simulated proofs from Σ_ϕ and τ_ϕ.

Lemma A.5. *For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ it is:*

$$Adv^{\mathcal{E}'}(Sim{+}^{\mathcal{E},\mathcal{S}_3}_{\mathcal{A}}(C, tx_{1:n}), Sim{+}^{\mathcal{E},\mathcal{S}_4}_{\mathcal{A}}(C, tx_{1:n})) = 0.$$

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

THE SIMULATOR $\mathcal{S}_5$ $\mathcal{S}_5$ is the same as $\mathcal{S}_4$, but instead of encrypting v in Line 5 of Alg. 3.2, $\mathcal{S}_5$ encrypts the constant 0.

Lemma A.6. *For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ the following advantage is negligible:*

$$Adv^{\mathcal{E}'}(Sim{+}^{\mathcal{E},\mathcal{S}_4}_{\mathcal{A}}(C, tx_{1:n}), Sim{+}^{\mathcal{E},\mathcal{S}_5}_{\mathcal{A}}(C, tx_{1:n})).$$

Proof. Follows from the IND-CPA property (Def. A.2) of the encryption scheme ($\mathcal{E}$ does not learn the secret key sk_{me} of the sender), and the fact that n is polynomial. Note that the introduced encryptions of 0 are never decrypted in $\text{Sim}{+}^{\mathcal{E},\mathcal{S}_5}_{\mathcal{A}}(C, tx_{1:n})$. Further, by the IND-CPA property, SimProof in $\mathcal{S}_4$ and $\mathcal{S}_5$ return indistinguishable proofs, even though its public input changes. □

THE SIMULATOR $\mathcal{S}_6$ $\mathcal{S}_6$ is the same as $\mathcal{S}_5$, but we change the behavior of Line 7 in Alg. 3.2 as follows.

First, the call to T_{plain} in the second case of Fig. 3.6, is replaced by the constant 0. That is, $\mathcal{S}_6$ simply computes $\text{Enc}(0, pk_{me}, r_i)$. Second, all calls to Enc_α in Fig. 3.8 for *honest* parties $\alpha \notin \mathcal{A}$ are replaced by fresh encryptions $\text{Enc}(0, pk_\alpha, r)$ of the constant 0.

Lemma A.7. *For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ the following advantage is negligible:*

$$Adv^{\mathcal{E}'}(Sim+_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_5}(C, tx_{1:n}), Sim+_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_6}(C, tx_{1:n})).$$

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

THE SIMULATOR $\mathcal{S}_7$ $\mathcal{S}_7$ is the same as $\mathcal{S}_6$, but we modify Eq. (3.22) as follows: If $\alpha \notin \mathcal{A}$, $\mathcal{S}_7$ computes $\text{Enc}(0, pk_\alpha, r)$.

Lemma A.8. *For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ the following advantage is negligible:*

$$Adv^{\mathcal{E}'}(Sim+_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_6}(C, tx_{1:n}), Sim+_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_7}(C, tx_{1:n})).$$

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

THE SIMULATOR $\mathcal{S}_8$ We finally define, for empty value $\perp$,

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

Lemma A.9. *For any PPT adversaries $\mathcal{E}, \mathcal{E}'$ it is:*

$$Adv^{\mathcal{E}'}(Sim+_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_7}(C, tx_{1:n}), Sim_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_8}(C, tx_{1:n})) = 0.$$

Proof. All calls to T_{plain} have been removed in $\mathcal{S}_7$, making rule (3.14) unreachable. Hence, $\mathcal{S}_7$ no longer accesses sk_{me}. Also, $\mathcal{S}_7$ no longer accesses $tx_{1:n}$. Therefore, the simulators $\mathcal{S}_7$ and $\mathcal{S}_8$ output the same distribution. □

A.4 EXPONENTIAL ELGAMAL ENCRYPTION

The ElGamal encryption scheme with messages in the exponent [48] is defined over a cyclic group G. It is IND-CPA (Def. A.2), assuming the decisional Diffie-Hellman assumption holds in G [137].

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

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

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

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

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

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

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

$$\begin{aligned} &\mathrm{Enc}(x, pk, r) \oplus \mathrm{Enc}(y, pk, r') = (g^{r+r'},\ g^{x+y} \cdot pk^{r+r'}) \\ &\quad = \mathrm{Enc}(x+y,\ pk,\ r+r'), \end{aligned} \tag{A.1}$$

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

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

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

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

A.5 CORRECTNESS OF ZAPPER

In the following, we assume that Zapper is instantiated with a *simulation-extractable zk-SNARK* (SE-SNARK [44]). This scheme satisfies perfect completeness [44, Def. 2.7], perfect zero-knowledge [44, Def. 2.8], and is simulation-extractable [44, Def. 2.10].

A.5.1 *Attacker Model*

Let $\mathcal{U}$ be the set of users in the system. In the following, we consider an active PPT adversary $\mathcal{E}$ which statically corrupts a subset $\mathcal{A} \subseteq \mathcal{U}$ of users, can create arbitrary transactions, and can observe and modify all transactions sent to the Zapper executor (formally defined shortly).

A.5.2 *Ideal World*

In Alg. A.2, we define a protocol $\mathcal{I}$ modeling the ideal world. This protocol maintains the current plaintext data of all objects in a *plain state* $\tilde{\sigma}$ mapping object ids to plain records. Additionally, it maintains a graph G_{key}, discussed next.

ACCESSIBLE OBJECTS A user u *can access* an object o if it knows the secret key of its owner. We use the directed graph G_{key} to formally specify the keys known by a given user. Specifically, the vertices in G_{key} represent public keys, and its edges connect $(\text{pk}_\alpha, \text{pk}_{\alpha'})$ if a user knowing sk_α also knows $\text{sk}_{\alpha'}$. Keys of (human) users represent the initial knowledge of the user and therefore have no incoming edges. Keys of objects o can be learned in two ways and therefore have two incoming edges. First, the user who created o also created its secret key, and therefore knows it, as indicated by edge $(\text{pk}_{me}, s.\text{pk}_{self})$ in Line 19. Second, any user knowing the secret key of the owner of o can decrypt o to learn its secret key, as indicated by edge $(s.\text{pk}_{owner}, s.\text{pk}_{self})$ in Line 20.

Overall, let $\text{pk}_{\mathcal{U}}$ denote the set of keys known by user u. Then, u can access object o, denoted by $\text{canAccess}_{G_{\text{key}}}(\text{pk}_{\mathcal{U}}, o)$ if G_{key} admits a path from a key in $\text{pk}_{\mathcal{U}}$ to the key of the owner of o. In this case, u can observe and interact with o in the ideal world.

INITIALIZATION In Alg. A.2, the function $\mathcal{I}$.Init accepts a set $\mathcal{C}$ of Zapper contracts, the identity of the dishonest users $\mathcal{A}$, and a map Pk mapping

Algorithm A.2 Ideal-world protocol $\mathcal{I}$.

```
 1: function Init($\mathcal{C}, \mathcal{A}, Pk$)
 2:     Remember $\mathcal{C}$
 3:     Initialize empty $G_{\text{key}}$ and $\tilde{\sigma}$
 4:     $pk_{\mathcal{A}} \leftarrow \{Pk[\alpha] \mid \alpha \in \mathcal{A}\}$        ▷ Public keys of dishonest users
 5:
 6: function Run($tx_{\text{ideal}}, \mathcal{T}, pk_{\text{acc}}, dryrun$)
 7:     if $tx_{\text{ideal}} = \perp \vee \neg\text{isUser}(tx_{\text{ideal}}.\text{pk}_{me})$ then return $\perp$
 8:     if values in $\mathcal{T}$ not unique then return $\perp$
 9:     $C.f, args, \text{pk}_{me} \leftarrow tx_{\text{ideal}}$
10:     run $C.f$ in $\mathcal{C}$ with arguments $args$ and sender address $\text{pk}_{me}$
11:       - use input state $\tilde{\sigma}$ and create output state $\tilde{\sigma}'$
          - for NEW, read new object id and object account secret key from $\mathcal{T}$
          - for FRESH, read fresh value from $\mathcal{T}$
          - collect sets $in$, $out$ and $new$ containing object ids of objects which are
          (i) accessed but not newly created, (ii) accessed but not destroyed, and
          (iii) newly created, resp., by $C.f$
12:     for $oid \in in$ do
13:         if $\neg\text{canAccess}_{G_{\text{key}}}(pk_{\text{acc}}, \tilde{\sigma}[oid])$ then
14:             return $\perp$
15:     $info_{\text{ideal}} \leftarrow$ GetInfo($C.f, pk_{\mathcal{A}}, G_{\text{key}}, \tilde{\sigma}, \tilde{\sigma}', in, out$)        ▷ Alg. A.4
16:     if $\neg dryrun$ then
17:         for each $oid \in new$ do
18:             $s \leftarrow \tilde{\sigma}'[oid]$
19:             Add edge $(\text{pk}_{me}, s.\text{pk}_{self})$ to $G_{\text{key}}$
20:             Add edge $(s.\text{pk}_{owner}, s.\text{pk}_{self})$ to $G_{\text{key}}$
21:         $\tilde{\sigma} \leftarrow \tilde{\sigma}'$
22:     return $info_{\text{ideal}}$
```

each user $u \in \mathcal{U}$ to its public key. It creates an empty plain state and key graph.

RUNNING TRANSACTIONS The function $\mathcal{I}$.Run accepts an *ideal transaction* tx_{ideal}, a *tape* $\mathcal{T}$ of unique values, a set pk_{acc} of public keys, and a boolean flag *dryrun*. Here, tx_{ideal} specifies the called function $C.f$, the function arguments *args* and the public key pk_{me} of the sender account used for the transaction (see Line 9). The tape $\mathcal{T}$ contains unique values to be used for the object ids and secret keys of new objects, and fresh values generated by **FRESH**. The set pk_{acc} is used for access checks: it contains the public keys corresponding to the secret keys known to the sender. Finally, the flag *dryrun* determines whether the tx_{ideal} should update the state.

$\mathcal{I}$.Run executes tx_{ideal} on the current plain state $\tilde{\sigma}$ using the values in $\mathcal{T}$ while keeping track of the current key graph G_{key}. For simplicity, we do not consider timestamps in the ideal world (Lines 10–11). Importantly, Lines 7–8 and Lines 12–14 abort by returning $\perp$ if the transaction is invalid (e.g., if pk_{acc} cannot access an input object). For the moment, the reader can ignore Line 15, which returns information visible to the adversary and will only be relevant for the privacy definition (App. A.7). Finally, Lines 16–21 update the state and G_{key} (discussed before).

A.5.3 *Real World*

The algorithm $\text{Real}^{\mathcal{E}}_{\mathcal{A}}$ (Alg. A.3, ignoring blue instructions) models the real world when executing a list $[tx_{\text{ideal}}]$ of transactions on Zapper classes $\mathcal{C}$. The reader can ignore Line 18 and all instructions highlighted in blue for the moment—they will be relevant later to formalize our notion of privacy.

ASSUMPTIONS Without loss of generality, we assume that all classes in $\mathcal{C}$ are registered at the assembly storage before the first transaction is submitted to the Zapper executor, and that user accounts are not shared. [1] Further, for simplicity, we assume that the Zapper executor runs on an idealized ledger, ignore the timestamp mechanism of Zapper, and assume that transactions of honest users are always created based on the latest object tree root hash β.

1 Sharing an account between two users u_1, u_2 is equivalent to introducing a new user u_3 with its own account, where u_3 is part of $\mathcal{A}$ iff u_1 or u_2 are in $\mathcal{A}$.

Algorithm A.3 $\text{Real}_{\mathcal{A}}^{\mathcal{E}}(\mathcal{C}, [tx_{\text{ideal}}])$ and $\text{Sim}_{\mathcal{A}}^{\mathcal{E},\mathcal{S}}(\mathcal{C}, [tx_{\text{ideal}}])$

```
1:  Trusted setup for zk-SNARK: $(\Sigma, \tau) \leftarrow \text{Setup}()$
2:  Register Zasm classes $\mathcal{C}$ at assembly storage
3:  For all $\alpha \in \mathcal{U}$, generate key pair $(\text{sk}_\alpha, \text{pk}_\alpha)$ and set $Pk[\alpha] \leftarrow \text{pk}_\alpha$
4:  Initialize empty Zapper system state $\sigma$
5:  $\mathcal{I}.\text{Init}(\mathcal{C}, \mathcal{A}, Pk)$
6:  $\mathcal{P}.\text{Init}(\mathcal{C}, \text{pk}, \{\text{sk}_\alpha\}_{\alpha \notin \mathcal{A}}, \Sigma)$        $\mathcal{S}.\text{Init}(\mathcal{C}, \text{pk}, \Sigma, \tau, \{\text{sk}_\alpha\}_{\alpha \in \mathcal{A}})$
7:  $\mathcal{E}.\text{Init}(\mathcal{C}, \text{pk}, \{\text{sk}_\alpha\}_{\alpha \in \mathcal{A}}, \Sigma)$
8:  for $tx_{\text{ideal}}$ in $[tx_{\text{ideal}}]$ do
9:      $tx \leftarrow \perp$
10:     if $tx_{\text{ideal}}.\text{pk}_{me} \in \{\text{pk}_\alpha\}_{\alpha \notin \mathcal{A}}$ then
11:         $info_{\text{ideal}} \leftarrow \mathcal{I}.\text{Run}(tx_{\text{ideal}}, \text{Rand}(), \{tx_{\text{ideal}}.\text{pk}_{me}\}, \text{true})$
12:         $tx \leftarrow \mathcal{P}.\text{Create}(tx_{\text{ideal}}, \sigma)$        $tx \leftarrow \mathcal{S}.\text{Create}(info_{\text{ideal}}, \sigma)$
13:     $tx' \leftarrow \mathcal{E}.\text{Create}(\sigma, tx)$
14:                                                  $\mathcal{S}.\text{Update}(tx', \mathcal{E})$
15:     $tx'_{\text{ideal}}, \mathcal{T}', pk_{\text{acc}} \leftarrow \text{Extract}(tx', \sigma, \Sigma)$
16:     $\mathcal{I}.\text{Run}(tx'_{\text{ideal}}, \mathcal{T}', pk_{\text{acc}}, \text{false})$
17:     Verify and execute $tx'$ on the current state $\sigma$
18: return $\mathcal{E}.\text{Decide}()$
```

MODELING USERS In $\text{Real}_{\mathcal{A}}^{\mathcal{E}}$, the steps of honest users are modeled by a protocol $\mathcal{P}$ which gets access to the honest users' secret keys (Line 6), where Line 12 creates transactions tx according to §4.5. The steps of the attacker are modeled by a PPT protocol $\mathcal{E}$, which gets access to the dishonest users' secret keys (Line 7) and can craft arbitrary transactions (Line 13). Importantly, the adversary gets access to transactions created by honest users in Line 13, allowing the adversary to observe and modify transactions before they are received at the Zapper executor.

TRACKING THE IDEAL WORLD In order to define correctness, we integrate the ideal-world protocol $\mathcal{I}$ in $\text{Real}_{\mathcal{A}}^{\mathcal{E}}$ (see highlighted). The ideal world is initialized in Line 5. In Line 11, ideal-world transactions tx_{ideal} of honest users are executed in dry-run mode in order to determine whether tx_{ideal} is valid and to obtain information visible to the adversary (relevant for privacy). Here, the new object ids, object secret keys, and fresh values in $\mathcal{T}$ are selected uniformly at random by Rand.

The ideal-world state is updated in Line 16. Here, an ideal-world transaction tx'_{ideal}, tape $\mathcal{T}'$ and keys pk_{acc} corresponding to the real-world transaction tx' are created in the Extract function as follows (Line 15): First, it is checked whether tx' is valid (i.e., whether according to σ, the proof Π is valid, the serial numbers are unique, and the unique seed is unique). If not, then Extract returns $(\bot, \bot, \bot)$. Otherwise, the witness extractor $\mathcal{X}$ of the SE-SNARK [44, Def. 2.10] is used to extract the private inputs $C'.f'$, $args$, pk_{me}, $[\text{sk}_\alpha]$ and $[R^{\text{pr}}]$ for ϕ from Π and (i) assemble the ideal-world transaction $tx'_{\text{ideal}} = (C'.f', args, \text{pk}_{me})$; (ii) compute $\mathcal{T}'$ from $[R^{\text{pr}}]$ by Lines 2–4 in Alg. 4.2; and (iii) derive $pk_{\text{acc}} = \{\text{derivePk}(\text{sk}_\alpha) \mid \text{sk}_\alpha \in [\text{sk}_\alpha]\}$.

A.5.4 *Correctness*

To define correctness, we finally introduce a function $\text{GetPlain}(\sigma, \{\text{sk}_\alpha\}_{\alpha \in \mathcal{U}})$, which computes an ideal-world plain state $\tilde{\sigma}$ corresponding to a real-world state σ as follows. First, it decrypts all records in the object tree of σ using the provided secret keys to obtain a set $\mathcal{R}$ of plain records. Next, it derives all serial numbers for all records in $\mathcal{R}$ and removes any records from $\mathcal{R}$ whose serial number appears in the serial number list of σ. The resulting plain records $\mathcal{R}$ are stored in a map $\tilde{\sigma}$ mapping object ids to plain records.

Theorem A.2 (Correctness of Zapper). *Assume Zapper is instantiated with an SE-SNARK [44]. Let $\mathcal{A}$ be any set of users, $\mathcal{C}$ a set of Zapper classes, $[tx_{ideal}]$ a list of n ideal-world transactions, where n is polynomial in the security parameter,* *and $\mathcal{E}$ a PPT protocol. Further, let σ be the state right after Line 17 during some iteration of the loop in $Real^{\mathcal{E}}_{\mathcal{A}}$, and $\tilde{\sigma}$ the ideal-world state held in $\mathcal{I}$ at this point. Then, with overwhelming probability, it is*

$$GetPlain(\sigma, \{sk_{\alpha}\}_{\alpha \in \mathcal{U}}) = \tilde{\sigma}.$$

Intuitively, Thm. A.2 captures the fact that for any transaction accepted in the real world, there exists a corresponding transaction accepted in the ideal world. In particular, transactions created or modified by the adversary must adhere to the Zasm code of registered classes.

Note that the ideal world only enforces new object ids, keys and fresh values to be unique (Line 8 in Alg. A.2). While honest users would compute these according to Lines 2–4 in Alg. 4.2 for uniform randomness, dishonest users may use non-random values for $[R^{\text{pr}}]$.

Proof of Thm. A.2. At a high level, Thm. A.2 follows from the simulation-extractability of the SE-SNARK and the construction of ϕ.

If tx' is rejected in the real world in Line 17, Extract returns $tx'_{\text{ideal}} = \perp$, which will also be rejected in the ideal world.

Otherwise, the proof Π in tx' is valid and Extract returns an ideal-world transaction $tx'_{\text{ideal}} = (C.f, args, \text{pk}_{me})$, tape $\mathcal{T}'$, and keys pk_{acc}.

First, we argue that $\mathcal{I}$.Run successfully updates its state and does not return $\perp$ when receiving $tx'_{\text{ideal}}, \mathcal{T}', pk_{\text{acc}}$. By the simulation-extractability of the SE-SNARK, all constraints in ϕ are satisfied with overwhelming probability. The check in Line 7 of Alg. A.2 succeeds as ϕ includes a corresponding constraint (Line 8 in Alg. 4.1). The check in Line 8 of Alg. A.2 succeeds with overwhelming probability by the fact that the entries in $\mathcal{T}'$ were computed according to Lines 2–4 in Alg. 4.2 and the collision-resistance of H_i (u in tx' is unique). Further, the keys pk_{acc} by construction allow to access all objects involved in the transaction, so the checks in Line 13 of Alg. A.2 succeed.

Second, we argue that the state update performed in $\mathcal{I}$.Run is reflected in the real world. By construction, the proof circuit ϕ ensures that the output records $[\hat{r}^{\text{out}}]$ included in tx' have been computed according to $C.f, args, \text{pk}_{me}$ based on *some* previous state of the involved objects. As the serial numbers in tx' are unique, the most recent state of all involved objects has been used. Therefore, with overwhelming probability, the plain records underlying $[\hat{r}^{\text{out}}]$ are equal to the corresponding records in the ideal-world state $\tilde{\sigma}$. □

A.6 AVAILABILITY AND INTEGRITY OF ZAPPER

In order to prove availability and integrity, we first formulate and prove some helper lemmas.

Lemma A.10 (Creation Success). *If $info_{ideal} \neq \bot$, then $\mathcal{P}$ succeeds to create a transaction $tx \neq \bot$ with valid proof Π in Line 12 of $Real_{\mathcal{A}}^{\mathcal{E}}$.*

Proof. By following §4.5, $\mathcal{P}$ succeeds to create a transaction with valid proof, provided $\mathcal{P}$ has access to (i) the sender's secret key sk_{me}, and (ii) the secret keys sk_{α} required to decrypt the input records.

For (i), $\mathcal{P}$ has learned sk_{me} in Line 6. For (ii), as $info_{\mathrm{ideal}} \neq \bot$, Line 13 in Alg. A.2 has confirmed that the user knowing pk_{me} can access all input objects. □

Lemma A.11 (Serial Nonce Uniqueness). *The serial nonces contained in any encrypted record stored in σ in Line 17 of $Real_{\mathcal{A}}^{\mathcal{E}}$ are globally unique with overwhelming probability.*

Proof. Transactions are only accepted in the real world if the unique seed u is globally unique. By the simulation-extractability of SE-SNARK, the serial nonces ρ_i^{out} contained in encrypted records of accepted transactions must have been computed according to Line 19 in Alg. 4.1 with overwhelming probability. The lemma thus follows from the collision-resistance of H_2. □

Lemma A.12 (Serial Number Uniqueness). *Assume $info_{ideal} \neq \bot$ in $Real_{\mathcal{A}}^{\mathcal{E}}$. Then, the serial numbers in tx computed in Line 12 cannot collide with any serial numbers in σ, except with negligible probability.*

Proof. By Lem. A.10, $\mathcal{P}$ successfully created tx, including a valid proof Π. Likewise, any transaction accepted in the real world included a valid proof. By the simulation-extractability of the SE-SNARK, any serial number sn_i in tx or σ has been computed according to Line 16 in Alg. 4.1. The corresponding serial nonce ρ_i^{in} either originates from (i) a non-dead input record or (ii) a dead input record introduced for padding. For case (i), ρ_i^{in} is unique with overwhelming probability by Lem. A.11. For case (ii), ρ_i^{in} is computed according to Line 15 in Alg. 4.1. By the uniqueness property of the construction Eq. (4.1) used in Eq. (4.4), it follows that ρ_i^{in} must be unique with overwhelming probability. The uniqueness of sn_i then follows from the collision-resistance of H_1 in Line 16 in Alg. 4.1. □

Theorem A.3 (Availability of Zapper). *Assume Zapper is instantiated with an SE-SNARK [44] and let $\mathcal{A}$, $\mathcal{C}$, $[tx_{ideal}]$, $\mathcal{E}$ as in Thm. A.2. In Line 17 of $Real_{\mathcal{A}}^{\mathcal{E}}$, the following holds with overwhelming probability:*

$$info_{ideal} \neq \perp \wedge tx = tx' \implies tx_{ideal} = tx'_{ideal}$$

Intuitively, Thm. A.3 ensures that if the adversary does not interfere and tx_{ideal} is valid in the ideal world, then tx' corresponds to tx_{ideal}. In particular, *previous* transactions of the adversary cannot prevent honest users from realizing any successful ideal-world transaction in the real world, thereby defeating the "Faerie Gold" attack [26].

Proof of Thm. A.3. Assume $info_{\text{ideal}} \neq \perp$ and $tx = tx'$. In this case, tx_{ideal} is valid in the ideal world and the adversary $\mathcal{E}$ did not modify the transaction tx originally created by $\mathcal{P}$.

First, we show that tx is accepted in the real world with overwhelming probability. By Lem. A.10, the proof in tx is valid. Further, the unique seed u in tx is indeed unique as it was selected by $\mathcal{P}$. Further, by Lem. A.12, all serial numbers in tx are unique with overwhelming probability.

Second, by the simulation-extractability of the SE-SNARK, the function Extract with overwhelming probability returns $tx'_{\text{ideal}} = tx_{\text{ideal}}$, as tx'_{ideal} is extracted from $tx' = tx$, where tx was created from tx_{ideal}. □

Theorem A.4 (Integrity of Zapper). *Assume Zapper is instantiated with an SE-SNARK [44] and let $\mathcal{A}$, $\mathcal{C}$, $[tx_{ideal}]$, $\mathcal{E}$ as in Thm. A.2. Consider Line 17 of $Real_{\mathcal{A}}^{\mathcal{E}}$ in any iteration. Let $\mathcal{O}$ be the set of objects accessed by tx'_{ideal}, and let Tx be the set of transactions tx created by $\mathcal{P}$ in Line 12 so far. The following holds with overwhelming probability:*

$$\begin{gathered} tx' \notin Tx \;\wedge\; tx'_{ideal} \neq \perp \\ \implies \\ tx'_{ideal}.pk_{me} \in pk_{\mathcal{A}} \;\wedge\; \forall o \in \mathcal{O}.\ \text{canAccess}_{G_{key}}(pk_{\mathcal{A}}, o). \end{gathered} \tag{A.2}$$

Further, let Tx' be the set of transactions tx' created by $\mathcal{E}$ in Line 13 in any previous iteration. Then:

$$tx' \in Tx' \implies tx'_{ideal} = \perp. \tag{A.3}$$

Intuitively, Eq. (A.2) captures the fact that the adversary can block or delay transactions by honest users, but cannot modify such transactions "in flight" such that they are still accepted, except if the resulting transactions could have been generated by the adversary from scratch. This for instance prevent attacks that change the arguments to a function call by an honest user in the attacker's favour. Further, Eq. (A.3) prevents replay attacks.

Proof of Thm. A.4. To prove Eq. (A.2), we assume $tx' \notin Tx$ and $tx'_{\text{ideal}} \neq \perp$.

As $tx'_{\text{ideal}} \neq \perp$, $tx'.\Pi$ is valid. Consider any $tx'' \in Tx$. As $tx' \notin Tx$, either $tx''.\Pi \neq tx'.\Pi$ or any of the public arguments used to generate $tx''.\Pi$ or $tx'.\Pi$, respectively, are different (all components $C.f, \beta, [sn], [\hat{r}^{\text{out}}], u$ of a transaction are public inputs of the proof circuit). Therefore, $\mathcal{E}$ has created a proof for a public input not seen previously, or a different proof for public inputs seen previously. By the simulation-extractability of the SE-SNARK, we can hence extract from $\mathcal{E}$ the private inputs sk_{me} and $[\text{sk}_\alpha]$ used to generate $tx'.\Pi$ with overwhelming probability. As $\mathcal{E}$ only has access to the secret keys corresponding to $\text{pk}^*_{\mathcal{A}}$, the theorem follows.

To prove Eq. (A.3), we observe that the serial numbers included in tx' are checked to be unique. Thus, repeated submissions of tx' are rejected. □

A.7 PRIVACY OF ZAPPER

We next formalize Zapper's privacy notion using a simulation-based definition.

SIMULATED REAL WORLD The algorithm $\text{Sim}^{\mathcal{E},\mathcal{S}}_{\mathcal{A}}$ (Alg. A.3, after replacing orange instructions by blue instructions) simulates the real world using a simulator protocol $\mathcal{S}$. In $\text{Sim}^{\mathcal{E},\mathcal{S}}_{\mathcal{A}}$, the steps of honest users are replaced by a simulator $\mathcal{S}$, which does not get access to the secret keys of honest users, but is provided the zk-SNARK simulation trapdoor τ and the secret keys of *dishonest* users (Line 6). Further, $\mathcal{S}$ gets access to the transactions crafted by the adversary as well as the internal state of $\mathcal{E}$ (Line 14). Transactions created by honest users are simulated by $\mathcal{S}$.Create based on information $info_{\text{ideal}}$ obtained from $\mathcal{I}$ (Line 12) as discussed next.

IDEAL-WORLD INFORMATION Alg. A.4 shows how the information $info_{\text{ideal}}$ visible to the users $\mathcal{A}$ in an ideal world is created as part of running an ideal-world transaction tx_{ideal} in Alg. A.2. The information $info_{\text{ideal}}$ includes (i) the name of the called function (see Line 2 in Alg. A.4); (ii) the object ids of all input objects the adversary can access (Line 5); and (iii) the plain state of all output objects the adversary can access (Line 10).

Algorithm A.4 Information visible to adversary in ideal world.

```
1:  function GetInfo($C.f, pk_\mathcal{A}, G_\text{key}, \tilde{\sigma}, \tilde{\sigma}', in, out$)
2:      $info_\text{ideal} \leftarrow$ {f: $C.f$, in: [], out: []}
3:      for each $oid \in in$ do
4:          if $\text{canAccess}_{G_\text{key}}(pk_\mathcal{A}, \tilde{\sigma}[oid])$ then
5:              $info_\text{ideal}$.in.append($oid$)
6:          else $info_\text{ideal}$.in.append(0)
7:      pad $info_\text{ideal}$.in by 0 to length $N_\text{obj}$
8:      for each $oid \in out$ do
9:          if $\text{canAccess}_{G_\text{key}}(pk_\mathcal{A}, \tilde{\sigma}[oid])$ then
10:             $info_\text{ideal}$.out.append($\tilde{\sigma}'[oid]$)
11:         else $info_\text{ideal}$.out.append(0)
12:     pad $info_\text{ideal}$.out by 0 to length $N_\text{obj}$
13:     return $info_\text{ideal}$
```

PRIVACY The goal of $\mathcal{E}$ is to decide whether it is interacting with the real world (Alg. A.3, orange) or a simulation thereof (Alg. A.3, blue) using the $\mathcal{E}$.Decide() function (Line 18 in Alg. A.3). If $\mathcal{E}$ cannot distinguish these two worlds except with negligible probability, then Zapper is private. We formalize this in Thm. A.5.

Theorem A.5 (Privacy of Zapper)**.** *Assume Zapper is instantiated with an SE-SNARK [44] and a key-private IK-CPA [115] and CPA-secure [138, Def. 3.22] encryption scheme. Let $\mathcal{A}$, $\mathcal{C}$, $[tx_{ideal}]$ as in Thm. A.2. There exists a PPT protocol $\mathcal{S}^\star$ such that for any PPT adversaries $\mathcal{E}$, $\mathcal{E}'$ the following advantage is negligible:*

$$Adv^{\mathcal{E}'}\left(Real^{\mathcal{E}}_{\mathcal{A}}(\mathcal{C}, [tx_{ideal}]), Sim^{\mathcal{E},\mathcal{S}^\star}_{\mathcal{A}}(\mathcal{C}, [tx_{ideal}])\right)$$

Proof. In the following, let $\mathcal{C}$, $\mathcal{A}$ and $[tx_\text{ideal}]$ as in Thm. A.5. We prove Thm. A.5 using a hybrid argument, by constructing a sequence PPT of simulators $\mathcal{S}_0, \ldots, \mathcal{S}_6$ and defining $\mathcal{S}^\star := \mathcal{S}_6$. Thm. A.5 follows from Lems. A.13–A.19 introduced below and the triangle inequality. □

SIMULATOR $\mathcal{S}_0$ We define $\text{Sim+}^{\mathcal{E},\mathcal{S}}_{\mathcal{A}}$ to be $\text{Sim}^{\mathcal{E},\mathcal{S}}_{\mathcal{A}}$ with the following modifications. First, $\mathcal{S}$.Init is additionally passed the secret keys of honest users $\{\text{sk}_\alpha\}_{\alpha \notin \mathcal{A}}$ in Line 6. Second, $\mathcal{S}$.Create is additionally passed tx_ideal in Line 12.

Next, we define $\mathcal{S}_0$ to forward the arguments received at Init and Create to the honest protocol $\mathcal{P}$ while ignoring the arguments τ, $\{\mathrm{sk}_\alpha\}_{\alpha\in\mathcal{A}}$ and $\mathit{info}_{\mathrm{ideal}}$. $\mathcal{S}_0$.Update does nothing.

Lemma A.13. *For any PPT protocols $\mathcal{E}$, $\mathcal{E}'$, the following advantage is zero:*

$$Adv^{\mathcal{E}'}\left(Real^{\mathcal{E}}_{\mathcal{A}}(\mathcal{C},[tx_{ideal}]), Sim+^{\mathcal{E},\mathcal{S}_0}_{\mathcal{A}}(\mathcal{C},[tx_{ideal}])\right)$$

Proof. By construction. □

SIMULATOR $\mathcal{S}_1$ $\mathcal{S}_1$ is the same as $\mathcal{S}_0$, but if $\mathit{info}_{\mathrm{ideal}} = \perp$, $\mathcal{S}_1$.Create does not forward tx_{ideal} to $\mathcal{P}$ but instead directly returns $\perp$. Otherwise, $\mathcal{S}_1$.Create ignores the zk-SNARK Π in the transaction returned by $\mathcal{P}$ and instead uses the proof simulator ZSimProve of the SE-SNARK [44, Def. 2.8] to create a simulated proof based on Σ and τ.

Lemma A.14. *For any PPT protocols $\mathcal{E}$, $\mathcal{E}'$, the following advantage is zero:*

$$Adv^{\mathcal{E}'}\left(Sim+^{\mathcal{E},\mathcal{S}_0}_{\mathcal{A}}(\mathcal{C},[tx_{ideal}]), Sim+^{\mathcal{E},\mathcal{S}_1}_{\mathcal{A}}(\mathcal{C},[tx_{ideal}])\right)$$

Proof. Follows from the perfect zero-knowledge property. □

SIMULATOR $\mathcal{S}_2$ $\mathcal{S}_2$ is the same as $\mathcal{S}_1$, but $\mathcal{S}_2$.Create ignores the function name $C.f$ and root hash β in the transaction returned by $\mathcal{P}$. Instead, $\mathcal{S}_2$ reads $C.f$ from the ideal-world information $\mathit{info}_{\mathrm{ideal}}$ and β from the current state σ.

Lemma A.15. *For any PPT protocols $\mathcal{E}$, $\mathcal{E}'$, the following advantage is zero:*

$$Adv^{\mathcal{E}'}\left(Sim+^{\mathcal{E},\mathcal{S}_1}_{\mathcal{A}}(\mathcal{C},[tx_{ideal}]), Sim+^{\mathcal{E},\mathcal{S}_2}_{\mathcal{A}}(\mathcal{C},[tx_{ideal}])\right)$$

Proof. By construction, the replaced values are equal. □

SIMULATOR $\mathcal{S}_3$ $\mathcal{S}_3$ is the same as $\mathcal{S}_2$, but $\mathcal{S}_3$.Create ignores the unique seed u in the transaction returned by $\mathcal{P}$ and instead selects a seed u uniformly at random.

Lemma A.16. *For any PPT protocols $\mathcal{E}$, $\mathcal{E}'$, the following advantage is zero:*

$$Adv^{\mathcal{E}'}\left(Sim+^{\mathcal{E},\mathcal{S}_2}_{\mathcal{A}}(\mathcal{C},[tx_{ideal}]), Sim+^{\mathcal{E},\mathcal{S}_3}_{\mathcal{A}}(\mathcal{C},[tx_{ideal}])\right)$$

Proof. Both $\mathcal{P}$ and $\mathcal{S}_3$ sample u from the same distribution. □

SIMULATOR $\mathcal{S}_4$ $\mathcal{S}_4$ simulates serial numbers. To this end, it keeps track of information known to the adversary. In particular, $\mathcal{S}_4$ is $\mathcal{S}_3$ with the following modifications.

First, $\mathcal{S}_4$ maintains a set *keys* and a map *data*. The set *keys* contains all secret keys known to $\mathcal{E}$. It consists of the secret keys $\{\mathrm{sk}_\alpha\}_{\alpha \in \mathcal{A}}$ of dishonest accounts and the secret keys of new objects created in transactions by $\mathcal{E}$. The map *data* maps object ids to a pair of serial nonce and secret key. In particular, *data* contains an entry for *oid* iff the most recent record of the object with id *oid* (i) was created by $\mathcal{E}$, and (ii) is encrypted under a public key pk_{owner} whose corresponding secret key sk_{owner} is known by $\mathcal{E}$. In this case, $data[oid]$ contains a pair $(\rho, \mathrm{sk}_{owner})$ containing the serial nonce ρ and owner account secret key sk_{owner} for the indicated object.

Next, to keep track of *keys* and *data*, $\mathcal{S}_4.\mathrm{Update}(tx', \mathcal{E})$ performs the following steps. First, if tx' has been previously simulated by $\mathcal{S}$.Create, the function returns. Otherwise, it verifies whether the zk-SNARK Π contained in tx' is valid. If no, the function returns. Otherwise, it uses the witness extractor $\mathcal{X}$ of the SE-SNARK [44, Def. 2.10] to extract from Π the secret keys of any newly created objects as part of *tx* and adds these to *keys*. Next, $\mathcal{S}_4$ uses the internal state of $\mathcal{E}$ to extract from Π: the owner public key pk^i_{owner}, object id oid_i, and serial nonce ρ^{out}_i of all non-dead output records r^{out}_i. For each i, if the secret key sk^i_{owner} corresponding to pk^i_{owner} is in *keys*, it updates $data[oid_i] \leftarrow (\rho^{\mathrm{out}}_i, \mathrm{sk}^i_{owner})$.

Finally, $\mathcal{S}_4$.Create ignores the serial numbers $[sn]$ in the transaction returned by $\mathcal{P}$ and instead simulates the i-th serial number by sn'_i as follows.

Case (i): If $oid := info_{\mathrm{ideal}}.\mathrm{in}[i] \neq 0$ and *data* contains an entry for *oid*, then $\mathcal{S}_4$ looks up $(\rho, \mathrm{sk}_{owner}) \leftarrow data[oid]$ and computes the serial number $sn'_i \leftarrow H_1(\rho \parallel \mathrm{sk}_{owner})$. Additionally, it deletes the entry $data[oid]$.

Case (ii):: Otherwise, $\mathcal{S}_4$ selects sn'_i uniformly at random.

Lemma A.17. *For any PPT protocols $\mathcal{E}$, $\mathcal{E}'$, the following advantage is negligible:*

$$Adv^{\mathcal{E}'}\left(Sim+^{\mathcal{E},\mathcal{S}_3}_{\mathcal{A}}(\mathcal{C}, [tx_{ideal}]), Sim+^{\mathcal{E},\mathcal{S}_4}_{\mathcal{A}}(\mathcal{C}, [tx_{ideal}])\right)$$

Proof. By construction, *keys* contains all secret keys known to the adversary $\mathcal{E}$. Because the owner of objects which have their own account cannot be changed, for every account α the adversary $\mathcal{E}$ either learns sk_α immediately or never.

For case (i), the serial number sn'_i is by construction identical to sn_i computed by $\mathcal{P}$.

For case (ii), the adversary by construction does not know either the secret key sk_{owner} or the nonce ρ_i^{in} used to derive sn_i in $\mathcal{P}$, or both. If the adversary does not know sk_{owner}, then sn_i' is indistinguishable from sn_i by the pseudorandomness property of H_1. Otherwise (i.e., the adversary does not know ρ_i^{in}), ρ_i^{in} must originate from $\mathcal{P}$, where we again distinguish two cases. (a) If the i-th input record is dead, then $\mathcal{P}$ has computed ρ_i^{in} according to Line 15 in Alg. 4.1 using fresh randomness. (b) Otherwise, ρ_i^{in} has been computed by $\mathcal{P}$ in a previous transaction according to Line 19 in Alg. 4.1 using fresh randomness.

In both cases (a) and (b), by the pseudorandomness property of H_2, sn_i is indistinguishable from a uniform random value. □

SIMULATOR $\mathcal{S}_5$ $\mathcal{S}_5$ simulates encrypted records. In particular, $\mathcal{S}_5$ is the same as $\mathcal{S}_4$, but all calls to $\mathcal{P}$ are removed and $info_{\text{ideal}}$ is used to simulate the i-th output record $\hat{r}_i^{\text{out}}$ by $\hat{r}_i'$ as follows.

Case (i): If $info_{\text{ideal}}.\text{out}[i] \neq 0$, $\mathcal{S}_5$ creates an encrypted record as follows. First, it selects a uniformly random serial nonce ρ'. Then, it selects uniform randomness R and creates $\hat{r}_i' = \mathrm{Enc}((\rho', s), s.\mathrm{pk}_{owner}, R)$, where $s = info_{\text{ideal}}.\text{out}[i]$.

Case (ii): Otherwise, $\mathcal{S}_5$ creates a fresh key pair $(\mathrm{sk}', \mathrm{pk}')$ and creates $\hat{r}_i' = \mathrm{Enc}((0,0), \mathrm{pk}', R)$ for uniform randomness R.

Lemma A.18. *For any PPT protocols $\mathcal{E}$, $\mathcal{E}'$, the following advantage is negligible:*

$$Adv^{\mathcal{E}'}\left(Sim{+}_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_4}(\mathcal{C}, [tx_{ideal}]), Sim{+}_{\mathcal{A}}^{\mathcal{E},\mathcal{S}_5}(\mathcal{C}, [tx_{ideal}])\right)$$

Proof. The transactions returned by $\mathcal{P}$.Create are completely ignored in $\mathcal{S}_5$, hence removing $\mathcal{P}$ does not change the distribution.

For case (i), the plaintext (ρ', s) encrypted by $\mathcal{S}_5$ is indistinguishable from the plaintext (ρ, r_i^{in}) in $\hat{r}_i^{\text{out}}$, except with negligible advantage: First, the nonce ρ has been derived by $\mathcal{P}$ according to Line 19 in Alg. 4.1 using fresh randomness, and the pseudorandomness of H_i ensures that ρ is indistinguishable from ρ'. Second, the values on the tape used in Line 11 are indistinguishable from the values created by $\mathcal{P}$ according to Lines 2–4 in Alg. 4.2 due to the pseudorandomness of H_i. Hence, by Thm. A.2, s is equal to r_i^{in} except with negligible probability.

For case (ii), $\mathcal{E}$ does not know the secret key which could be used to decrypt $\hat{r}_i^{\text{out}}$ produced by $\mathcal{P}$ (note that any dead records are encrypted using the dishonest sender account's key). By the IK-CPA property and

CPA-security of the encryption scheme, $\hat{r}'_i$ is indistinguishable from $\hat{r}_i^{\text{out}}$ except with negligible advantage. □

SIMULATOR $\mathcal{S}_6$ We define $\mathcal{S}_6$ as follows:

$$\mathcal{S}_6.\text{Init}(\mathcal{C},\ldots,\{\text{sk}_\alpha\}_{\alpha\in\mathcal{A}}) := \mathcal{S}_5.\text{Init}(\mathcal{C},\ldots,\{\text{sk}_\alpha\}_{\alpha\in\mathcal{A}},\emptyset)$$
$$\mathcal{S}_6.\text{Create}(\mathit{info}_{\text{ideal}},\sigma) := \mathcal{S}_5.\text{Create}(\mathit{info}_{\text{ideal}},\sigma,\emptyset)$$

Lemma A.19. *For any PPT protocols $\mathcal{E}$, $\mathcal{E}'$, the following advantage is zero:*

$$Adv^{\mathcal{E}'}\left(Sim+^{\mathcal{E},\mathcal{S}_5}_{\mathcal{A}}(\mathcal{C},[tx_{ideal}]),Sim+^{\mathcal{E},\mathcal{S}_6}_{\mathcal{A}}(\mathcal{C},[tx_{ideal}])\right)$$

Proof. $\mathcal{S}_5$ no longer accesses $\{\text{sk}_\alpha\}_{\alpha\notin\mathcal{A}}$ or tx_{ideal}, hence the simulators $\mathcal{S}_6$ and $\mathcal{S}_5$ output same distribution. □

BIBLIOGRAPHY

[1] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. "zkay: Specifying and Enforcing Data Privacy in Smart Contracts". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019. DOI: 10.1145/3319535.3363222.

[2] Samuel Steffen, Benjamin Bichsel, Roger Baumgartner, and Martin Vechev. "ZeeStar: Private Smart Contracts by Homomorphic Encryption and Zero-knowledge Proofs". In: *2022 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2022. DOI: 10.1109/SP46214.2022.9833732.

[3] Samuel Steffen, Benjamin Bichsel, and Martin Vechev. "Zapper: Smart Contracts with Data and Identity Privacy". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2022. DOI: 10.1145/3548606.3560622.

[4] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. "Unsupervised learning of API aliasing specifications". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2019. DOI: 10.1145/3314221.3314640.

[5] Timon Gehr, Samuel Steffen, and Martin Vechev. "λPSI: exact inference for higher-order probabilistic programs". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2020. DOI: 10.1145/3385412.3386006.

[6] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. "Probabilistic Verification of Network Configurations". In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. ACM, 2020. DOI: 10.1145/3387514.3405900.

[7] Benjamin Bichsel, Samuel Steffen, Ilija Bogunovic, and Martin Vechev. "DP-Sniper: Black-Box Discovery of Differential Privacy Violations using Classifiers". In: *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021. DOI: 10.1109/SP40001.2021.00081.

[8] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. "Unqomp: Synthesizing Uncomputation in Quantum Circuits". In: *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2021. DOI: 10.1145/3453483.3454040.

[9] Nikola Jovanović, Marc Fischer, Samuel Steffen, and Martin Vechev. "Private and Reliable Neural Network Inference". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2022. DOI: 10.1145/3548606.3560709.

[10] Fabian Vogelsteller and Vitalik Buterin. *EIP-20: Token Standard*. Ethereum Improvement Proposals. https://eips.ethereum.org/EIPS/eip-20. Accessed: 2022-09-07. 2015.

[11] Unsiwap Labs. *Uniswap decentralized trading protocol*. https://uniswap.org/. Accessed: 2021-07-14.

[12] CryptoGames. *Crypto.Games Ethereum Lottery*. https://crypto.games/lottery/ethereum. Accessed: 2022-08-15.

[13] Muyao Shen. *Crypto Valley Declares Blockchain Voting Trial a 'Success'*. https://www.coindesk.com/crypto-valley-declares-blockchain-voting-trial-a-success/. Accessed: 2022-09-07. 2018.

[14] Lydia Torne and Sophie Sheldon. *Medical data and the rise of blockchain*. http://www.pharmatimes.com/web_exclusives/medical_data_and_the_rise_of_blockchain_1243441. Accessed: 2022-09-07. 2018.

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

[16] Gavin Wood. *Ethereum: a Secure Decentralised Generalised Transaction Ledger*. https://gavwood.com/paper.pdf. Accessed: 2022-09-07. 2016.

[17] Evan Duffield and Daniel Diaz. *Dash: A Privacy Centric Crypto Currency*. https://blockchainlab.com/pdf/Dash-WhitepaperV1.pdf. Accessed: 2022-09-07. 2014.

[18] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A. Kroll, and Edward W. Felten. "Mixcoin: Anonymity for Bitcoin with Accountable Mixes". In: *Financial Cryptography and Data Security*. Springer, 2014. DOI: 10.1007/978-3-662-45472-5_31.

[19] Dimaz Ankaa Wijaya, Joseph K. Liu, Ron Steinfeld, Shi-Feng Sun, and Xinyi Huang. "Anonymizing Bitcoin Transaction". In: *Information Security Practice and Experience*. Springer, 2016. DOI: 10.1007/978-3-319-49151-6_19.

[20] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. "CoinShuffle: Practical Decentralized Coin Mixing for Bitcoin". In: *Computer Security - ESORICS*. Springer, 2014. DOI: 10.1007/978-3-319-11212-1_20.

[21] Jan Henrik Ziegeldorf, Roman Matzutt, Martin Henze, Fred Grossmann, and Klaus Wehrle. "Secure and anonymous decentralized Bitcoin mixing". In: *Future Generation Computer Systems* 80 (2018). DOI: 10.1016/j.future.2016.05.018.

[22] Niluka Amarasinghe, Xavier Boyen, and Matthew McKague. "A Survey of Anonymity of Cryptocurrencies". In: *Proceedings of the Australasian Computer Science Week Multiconference (ACSW)*. ACM, 2019. DOI: 10.1145/3290688.3290693.

[23] Shen Noether. *Ring Signature Confidential Transactions for Monero*. Cryptology ePrint Archive, Paper 2015/1098. https://eprint.iacr.org/2015/1098. 2015.

[24] I. Miers, C. Garman, M. Green, and A. D. Rubin. "Zerocoin: Anonymous Distributed E-Cash from Bitcoin". In: *2013 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2013. DOI: 10.1109/SP.2013.34.

[25] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. "Zerocash: Decentralized Anonymous Payments from Bitcoin". In: *2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014. DOI: 10.1109/SP.2014.36.

[26] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. *Zcash Protocol Specification*. https://zips.z.cash/protocol/protocol.pdf. Accessed: 2022-09-07. 2021.

[27] Antoine Rondelet and Michal Zajac. "ZETH: On Integrating Zerocash on Ethereum". In: *arXiv:1904.00905 [cs]* (2019). URL: http://arxiv.org/abs/1904.00905.

[28] Zhangshuang Guan, Zhiguo Wan, Yang Yang, Yan Zhou, and Butian Huang. "BlockMaze: An Efficient Privacy-Preserving Account-Model Blockchain Based on zk-SNARKs". In: *IEEE Transactions on Dependable and Secure Computing* (2020). DOI: 10.1109/TDSC.2020.3025129.

[29] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. "Zether: Towards Privacy in a Smart Contract World". In: *Financial Cryptography and Data Security*. Springer, 2020. DOI: 10.1007/978-3-030-51280-4_23.

[30] Benjamin E. Diamond. "Many-out-of-Many Proofs and Applications to Anonymous Zether". In: *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021. DOI: 10.1109/SP40001.2021.00026.

[31] Prastudy Fauzi, Sarah Meiklejohn, Rebekah Mercer, and Claudio Orlandi. "Quisquis: A New Design for Anonymous Cryptocurrencies". In: *Advances in Cryptology (ASIACRYPT)*. Springer, 2019. DOI: 10.1007/978-3-030-34578-5_23.

[32] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts". In: *2016 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016. DOI: 10.1109/SP.2016.55.

[33] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. "Arbitrum: Scalable, private smart contracts". In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner.

[34] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019. DOI: 10.1109/EuroSP.2019.00023.

[35] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. "FastKitten: Practical Smart Contracts on Bitcoin". In: *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/das.

[36] Aritra Banerjee, Michael Clear, and Hitesh Tewari. "zkHawk: Practical Private Smart Contracts from MPC-based Hawk". In: *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. 2021. DOI: 10.1109/BRAINS52497.2021.9569822.

[37] Aritra Banerjee and Hitesh Tewari. *Multiverse of HawkNess: A Universally-Composable MPC-based Hawk Variant*. Cryptology ePrint Archive, Paper 2022/421. https://eprint.iacr.org/2022/421. 2022.

[38] Ravital Solomon and Ghada Almashaqbeh. *smartFHE: Privacy-Preserving Smart Contracts from Fully Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2021/133. https://eprint.iacr.org/2021/133. 2021.

[39] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. "ZEXE: Enabling Decentralized Private Computation". In: *2020 IEEE Symposium on Security and Privacy (S&P)*. ISSN: 2375-1207. IEEE, 2020. DOI: 10.1109/SP40000.2020.00050.

[40] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, and Nicolas Christin. "An Empirical Analysis of Traceability in the Monero Blockchain". In: *Proceedings on Privacy enhancing Technologies (PoPETs)* (2018). DOI: 10.1515/popets-2018-0025.

[41] Manuel Blum, Paul Feldman, and Silvio Micali. "Non-interactive zero-knowledge and its applications". In: *Proceedings of the twentieth annual ACM symposium on Theory of computing (STOC)*. ACM, 1988. DOI: 10.1145/62212.62222.

[42] Uriel Feige, Dror Lapidot, and Adi Shamir. "Multiple Non-Interactive Zero Knowledge Proofs Under General Assumptions". In: *SIAM J. Comput.* 29.1 (1999). DOI: 10.1137/S0097539792230010.

[43] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again". In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS)*. ACM, 2012. DOI: 10.1145/2090236.2090263.

[44] Jens Groth and Mary Maller. "Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs". In: *Advances in Cryptology (CRYPTO)*. Springer, 2017. DOI: 10.1007/978-3-319-63715-0_20.

[45] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *Advances in Cryptology (EUROCRYPT)*. Springer, 2016. DOI: 10.1007/978-3-662-49896-5_11.

[46] Jacob Eberhardt and Stefan Tai. "ZoKrates - Scalable Privacy-Preserving Off-Chain Computations". In: *IEEE International Conference on Blockchain*. IEEE, 2018. DOI: 10.1109/Cybermatics_2018.2018.00199.

[47] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology (EUROCRYPT)*. Springer, 1999. DOI: 10.1007/3-540-48910-X_16.

[48] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. "A secure and optimally efficient multi-authority election scheme". In: *European Transactions on Telecommunications* 8.5 (1997). DOI: 10.1002/ett.4460080506.

[49] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO)*. Springer, 1987. DOI: 10.1007/3-540-48184-2_32.

[50] B. Laurie, A. Langley, and E. Kasper. *Certificate Transparency*. RFC 6962. 2013.

[51] Vitalik Buterin. *Privacy on the Blockchain*. https://blog.ethereum.org/2016/01/15/privacy-on-the-blockchain/. Accessed: 2022-09-07. 2016.

[52] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. "Quadratic Span Programs and Succinct NIZKs without PCPs". In: *Advances in Cryptology (EUROCRYPT)*. Springer, 2013. DOI: 10.1007/978-3-642-38348-9_37.

[53] B. Parno, J. Howell, C. Gentry, and M. Raykova. "Pinocchio: Nearly Practical Verifiable Computation". In: *2013 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2013. DOI: 10.1109/SP.2013.47.

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

[55] Vitalik Buterin and Christian Reitwiessner. *EIP-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128*. Ethereum Improvement Proposals. https://eips.ethereum.org/EIPS/eip-197. Accessed: 2022-09-07. 2017.

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

[57] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*. ACM, 1977. DOI: `10.1145/512950.512973`.

[58] Michael Backes and Dominique Unruh. “Computational Soundness of Symbolic Zero-Knowledge Proofs Against Active Attackers”. In: *2008 IEEE 21st Computer Security Foundations Symposium (CSF)*. IEEE, 2008. DOI: `10.1109/CSF.2008.20`.

[59] D. Dolev and A. C. Yao. “On the security of public key protocols”. In: *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science (SFCS)*. IEEE, 1981. DOI: `10.1109/SFCS.1981.32`.

[60] David Basin, Jannik Dreier, and Ralf Sasse. “Automated Symbolic Proofs of Observational Equivalence”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2015. DOI: `10.1145/2810103.2813662`.

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

[62] Nick Baumann, Samuel Steffen, Benjamin Bichsel, Petar Tsankov, and Martin Vechev. “zkay v0.2: Practical Data Privacy for Smart Contracts”. In: *arXiv:2009.01020 [cs]* (2020). Technical Report. URL: `https://arxiv.org/abs/2009.01020`.

[63] ConsenSys. *Truffle Suite*. `https://trufflesuite.com/`. Accessed: 2022-09-07. 2018.

[64] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. “Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs”. In: *2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015. DOI: `10.1109/SP.2015.25`.

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

[66] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. “Securify: Practical Security Analysis of Smart Contracts”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018. DOI: `10.1145/3243734.3243780`.

[67] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. "KEVM: A Complete Semantics of the Ethereum Virtual Machine". In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018. DOI: 10.1109/CSF.2018.00022.

[68] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution". In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[69] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. "An End-to-End System for Large Scale P2P MPC-as-a-Service and Low-Bandwidth MPC for Weak Participants". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018. DOI: 10.1145/3243734.3243801.

[70] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. "Global-Scale Secure Multiparty Computation". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017. DOI: 10.1145/3133956.3133979.

[71] Janus Dam Nielsen and Michael I. Schwartzbach. "A domain-specific programming language for secure multiparty computation". In: *Proceedings of the 2007 workshop on Programming languages and analysis for security (PLAS)*. ACM, 2007. DOI: 10.1145/1255329.1255333.

[72] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. "Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations". In: *2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014. DOI: 10.1109/SP.2014.48.

[73] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. "Automating Efficient RAM-Model Secure Computation". In: *2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014. DOI: 10.1109/SP.2014.46.

[74] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. "ObliVM: A Programming Framework for Secure Computation". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015. DOI: 10.1109/SP.2015.29.

[75] Dan Bogdanov, Peeter Laud, and Jaak Randmets. "Domain-Polymorphic Programming of Privacy-Preserving Applications". In: *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, 2014. DOI: 10.1145/2637113.2637119.

[76] Daniel Demmler, Thomas Schneider, and Michael Zohner. "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation". In: *Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2015. DOI: 10.14722/ndss.2015.23113.

[77] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. "ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/patra.

[78] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. "Viaduct: An Extensible, Optimizing Compiler for Secure Distributed Programs". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2021. DOI: 10.1145/3453483.3454074.

[79] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. "Pinocchio: Nearly Practical Verifiable Computation". In: *2013 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2013. DOI: 10.1109/SP.2013.47.

[80] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. "Geppetto: Versatile Verifiable Computation". In: *2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015. DOI: 10.1109/SP.2015.23.

[81] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. "Efficient RAM and Control Flow in Verifiable Outsourced Computation". In: *Proceedings 2015 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2015. DOI: 10.14722/ndss.2015.23097.

[82] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. "SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge". In: *Advances in Cryptology (CRYPTO)*. Springer, 2013. DOI: 10.1007/978-3-642-40084-1_6.

[83] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. "Succinct Non-interactive Zero Knowledge for a Von Neumann Architecture". In: *Proceedings of the 23rd USENIX conference on Security Symposium (SEC)*. USENIX Association, 2014. DOI: 10.5555/2671225.2671275.

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

[85] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. "xJsnark: A Framework for Efficient Verifiable Computation". In: *2018 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2018. DOI: 10.1109/SP.2018.00018.

[86] Andrew C. Myers. "JFlow: Practical Mostly-static Information Flow Control". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 1999. DOI: 10.1145/292540.292561.

[87] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. "A Language for Automatically Enforcing Privacy Policies". In: *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM, 2012. DOI: 10.1145/2103656.2103669.

[88] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. "Faceted Execution of Policy-agnostic Programs". In: *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security (PLAS)*. ACM, 2013. DOI: 10.1145/2465106.2465121.

[89] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. "Precise, Dynamic Information Flow for Database-backed Applications". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2016. DOI: 10.1145/2908080.2908098.

[90] Monika di Angelo and Gernot Salzer. "Characterizing Types of Smart Contracts in the Ethereum Landscape". In: *Financial Cryptography and Data Security*. Springer, 2020. DOI: 10.1007/978-3-030-54455-3_28.

[91] ConsenSys. *Ethereum Smart Contract Best Practices: Denial of Service*. https://consensys.github.io/smart-contract-best-practices/attacks/denial-of-service/. Accessed: 2022-07-28. 2022.

[92] Paulo S. L. M. Barreto and Michael Naehrig. “Pairing-Friendly Elliptic Curves of Prime Order”. In: *Selected Areas in Cryptography*. Springer, 2006. DOI: `10.1007/11693383_22`.

[93] Barry WhiteHat, Jordi Baylina, and Marta Bellés. *Baby Jubjub Elliptic Curve*. `https://iden3-docs.readthedocs.io/en/latest/iden3_repos/research/publications/zkproof-standards-workshop-2/baby-jubjub/baby-jubjub.html`. Accessed: 2022-09-07. 2019.

[94] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. “Bulletproofs: Short Proofs for Confidential Transactions and More”. In: *2018 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2018. DOI: `10.1109/SP.2018.00020`.

[95] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. “Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2019. DOI: `10.1145/3319535.3339817`.

[96] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. “Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation”. In: *Advances in Cryptology (CRYPTO)*. Springer, 2019. DOI: `10.1007/978-3-030-26954-8_24`.

[97] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Paper 2019/953. `https://eprint.iacr.org/2019/953`. 2019.

[98] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. “Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS”. In: *Advances in Cryptology (EUROCRYPT)*. Springer, 2020. DOI: `10.1007/978-3-030-45721-1_26`.

[99] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. “Transparent SNARKs from DARK Compilers”. In: *Advances in Cryptology (EUROCRYPT)*. Springer, 2020, 677.

[100] D. Shanks. “Class number, a theory of factorization, and genera”. In: *Proc. Symp. Pure Math*. Vol. 20. American Mathematical Society, 1971.

[101] Ahmed Kosba. *jsnark: A Java library for zk-SNARK circuits*. `https://github.com/akosba/jsnark`. Accessed: 2021-08-11.

[102] SCIPR Lab and contributors. *libsnark: a C++ library for zkSNARK proofs*. https://github.com/scipr-lab/libsnark. Accessed: 2021-08-11.

[103] Shimon Even, Oded Goldreich, and Abraham Lempel. "A Randomized Protocol for Signing Contracts". In: *Commun. ACM* 28.6 (1985). DOI: 10.1145/3812.3818.

[104] Jacob Rozen. *The ridiculously high cost of Gas on Ethereum*. CoinGeek Editorial. https://coingeek.com/the-ridiculously-high-cost-of-gas-on-ethereum/. Accessed: 2021-07-14. 2021.

[105] Youssef Faqir-Rhazoui, Miller-Janny Ariza-Garzón, Javier Arroyo, and Samer Hassan. "Effect of the Gas Price Surges on User Activity in the DAOs of the Ethereum Blockchain". In: *Conference on Human Factors in Computing Systems (CHI)*. ACM, 2021. DOI: 10.1145/3411763.3451755.

[106] ethereum.org contributors. *The Eth2 vision*. https://ethereum.org/en/eth2/vision/. Accessed: 2021-07-14. 2021.

[107] Shunli Ma, Yi Deng, Debiao He, Jiang Zhang, and Xiang Xie. "An Efficient NIZK Scheme for Privacy-Preserving Transactions Over Account-Model Blockchain". In: *IEEE Transactions on Dependable and Secure Computing* 18.2 (2021). DOI: 10.1109/TDSC.2020.2969418.

[108] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. "SoK: Fully Homomorphic Encryption Compilers". In: *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021. DOI: 10.1109/SP40001.2021.00068.

[109] Jiwon Lee, Jaekyoung Choi, Jihye Kim, and Hyunok Oh. *SAVER: SNARK-friendly, Additively-homomorphic, and Verifiable Encryption and decryption with Rerandomization*. Cryptology ePrint Archive, Oaoer 2019/1270. https://eprint.iacr.org/2019/1270. 2019.

[110] Fergal Reid and Martin Harrigan. "An Analysis of Anonymity in the Bitcoin System". In: *Security and Privacy in Social Networks*. Springer, 2013. DOI: 10.1007/978-1-4614-4139-7_10.

[111] Dave Jevans. *CipherTrace Files Two Monero Cryptocurrency Tracing Patents*. https://ciphertrace.com/ciphertrace-files-two-monero-cryptocurrency-tracing-patents/. Accessed: 2022-04-15. 2020.

[112] Wolfie Zhao. *Bithumb $31 Million Crypto Exchange Hack: What We Know (And Don't)*. https://www.coindesk.com/markets/2018/06/20/bithumb-31-million-crypto-exchange-hack-what-we-know-and-dont/. Accessed: 2022-04-20. 2018.

[113] Andrew Norry. *The History of the Mt Gox Hack: Bitcoin's Biggest Heist*. Blockonomi. https://blockonomi.com/mt-gox-hack/. Accessed: 2022-04-20. 2020.

[114] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. "SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain". In: *Financial Cryptography and Data Security*. Springer, 2020. DOI: 10.1007/978-3-030-43725-1_13.

[115] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. "Key-Privacy in Public-Key Encryption". In: *Advances in Cryptology (ASIACRYPT)*. Springer, 2001. DOI: 10.1007/3-540-45682-1_33.

[116] Roger Dingledine, Nick Mathewson, and Paul Syverson. "Tor: The Second-Generation Onion Router". In: *13th USENIX Security Symposium (USENIX Security 04)*. USENIX Association, 2004. URL: https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router.

[117] Arkworks Contributors. *arkworks zkSNARK ecosystem*. https://arkworks.rs. Accessed: 2022-09-07. 2022.

[118] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. "Constructing Elliptic Curves with Prescribed Embedding Degrees". In: *Security in Communication Networks*. Springer, 2003. DOI: 10.1007/3-540-36413-7_19.

[119] Sean Bowe. *Jujub elliptic curve*. https://github.com/zkcrypto/jubjub. Accessed: 2022-09-07. 2017.

[120] S. Micali, M. Rabin, and J. Kilian. "Zero-knowledge sets". In: *44th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 2003. DOI: 10.1109/SFCS.2003.1238183.

[121] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. "BLAKE2: Simpler, Smaller, Fast as MD5". In: *Applied Cryptography and Network Security*. Springer, 2013. DOI: 10.1007/978-3-642-38980-1_8.

[122] Toghrul Maharramov, Dmitry Khovratovich, and Emanuele Francioni. *The Dusk Network Whitepaper*. https://dusk.network/uploads/The_Dusk_Network_Whitepaper_v3_0_0.pdf. Accessed: 2022-04-28. 2021.

[123] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. "Poseidon: A New Hash Function for Zero-Knowledge Proof Systems". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/grassi.

[124] T. Elgamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *IEEE Transactions on Information Theory* 31.4 (1985). DOI: 10.1109/TIT.1985.1057074.

[125] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications". In: *Selected Areas in Cryptography*. Springer, 2012. DOI: 10.1007/978-3-642-28496-0_19.

[126] Dmitry Khovratovich. *Encryption with Poseidion*. https://dusk.network/uploads/Encryption-with-Poseidon.pdf. Accessed: 2022-09-07. 2019.

[127] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. *ZEXE: Enabling Decentralized Private Computation*. Cryptology ePrint Archive, Paper 2018/962 (updated 2021-03-30). https://eprint.iacr.org/2018/962. 2021.

[128] StarkNet. *StarkNet Main Webpage*. https://starknet.io/. Accessed: 2022-05-01. 2022.

[129] zkSync. *zkSync Main Webpage*. https://zksync.io/. Accessed: 2022-05-01. 2022.

[130] Zachary Williamson. *The AZTEC Protocol*. https://github.com/AztecProtocol/AZTEC/blob/master/AZTEC.pdf. Accessed: 2022-05-01. 2021.

[131] StarkNet. *Privacy on StarkNet*. https://starknet.io/faq/privacy-on-starknet/. Accessed: 2022-05-01. 2022.

[132] zkSync. *zkSync Documentation: Privacy*. https://docs.zksync.io/userdocs/privacy.html. Accessed: 2022-05-01. 2022.

[133] Zac Williamson. *Aztec's ZK-ZK-Rollup, looking behind the cryptocurtain*. Medium. `https://medium.com/aztec-protocol/aztecs-zk-zk-rollup-looking-behind-the-cryptocurtain-2b8af1fca619`. Accessed: 2022-05-01. 2021.

[134] David Heath and Vladimir Kolesnikov. "A 2.1 KHz Zero-Knowledge Processor with BubbleRAM". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2020. DOI: `10.1145/3372297.3417283`.

[135] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. "Zero Knowledge for Everything and Everyone: Fast ZK Processor with Cached ORAM for ANSI C Programs". In: *2021 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021. DOI: `10.1109/SP40001.2021.00089`.

[136] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. "Constant-Overhead Zero-Knowledge for RAM Programs". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2021. DOI: `10.1145/3460120.3484800`.

[137] Yiannis Tsiounis and Moti Yung. "On the security of ElGamal based encryption". In: *Public Key Cryptography*. Springer, 1998. DOI: `10.1007/BFb0054019`.

[138] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, 2007.