ABSTRACT

Title of dissertation: VERIFIABLE COMPUTATION IN PRACTICE: TOOLS AND PROTOCOLS

Ahmed Kosba
Doctor of Philosophy, 2018

Dissertation directed by: Prof. Charalampos Papamanthou, Prof. Elaine Shi
Department of Computer Science

Verifiable computation (VC) protocols enable clients to outsource computations to untrusted servers in the cloud without compromising the integrity of the computation. Although cryptographic approaches for verifiable computation were mostly of theoretical interest in the past, there has been great progress in the area during the past few years. In particular, efficient constructions for Zero-Knowledge Succinct Non-interactive ARguments of Knowledge (zk-SNARKs) were proposed and adopted in practice. These techniques enable an untrusted server to prove the correctness of computations in zero-knowledge using a succinct proof that can be verified efficiently by the client.

This thesis aims at addressing some challenges in such VC protocols, and developing practical protocols for cryptocurrency applications. The challenges we address include the proof computation overhead at the prover's side, and the level of expertise expected from the programmers to write secure and efficient programs for VC. More specifically, current protocols require the programmer to carefully

express the computation as an arithmetic circuit, in a way that minimizes the proof computation overhead and prevents malicious behavior by the prover, which is a non-trivial task.

To address the above challenges, we present a framework that aims to reduce the proof computation overhead, and offer more programmability to non-specialist developers, while automating the task of circuit minimization through a combination of techniques. The framework includes new circuit-friendly algorithms for frequent operations, which achieve constant to asymptotic savings over algorithms used in previous compilers. In addition, we explore and optimize cryptographic primitives that have efficient arithmetic circuit representations.

Furthermore, we explore different settings where VC can be used in practice. We present the design of Hawk, a system for privacy-preserving smart contracts. Hawk enables custom decentralized applications in the smart contract setting to run verifiably on top of a public blockchain system, while not revealing the participants' inputs to the network. To achieve practical performance, Hawk relies on a special party per contract (a manager) that is only trusted for posterior privacy, but not for correctness. Finally, we explore how VC techniques and smart contracts could enable practical crimes in the future, which highlights the importance of working on countermeasures.

Verifiable Computation in Practice: Tools and Protocols

by

Ahmed Kosba

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2018

Advisory Committee:
Prof. Charalampos Papamanthou, Chair/Co-Advisor
Prof. Elaine Shi, Co-Chair/Co-Advisor
Prof. Hector Corrada Bravo
Prof. Jonathan Katz
Prof. Lawrence Washington

Acknowledgments

First, I would like to thank my advisors Babis Papamanthou and Elaine Shi for their help and guidance during the past five years. I am grateful for their support and the opportunities they provided me. Taking Elaine's class in the very beginning sparked my interest in doing research in the area, and her enthusiasm was always inspiring. I also had the pleasure of working with Babis most of the time during my PhD, with whom I spent hours thinking and brainstorming about problems in long meetings, for which he was always available.

I also would like to thank my collaborators during my PhD studies, most notably Andrew Miller and Ari Juels, with whom some work in this thesis was co-authored. I am also grateful to professors Hector Corrada Bravo, Jonathan Katz and Lawrence Washington for being on my dissertation committee.

Studying at the Computer Science Department and the Maryland Cybersecurity Center (MC2) at UMD has been a pleasure. I had the opportunity to learn from great professors and meet wonderful students. I am also thankful to the administrative staff for making a lot of things easier for the students.

Finally, I do not think words can describe the gratitude I owe for my parents, who went above and beyond to help with and support my education since I was young. I am also very grateful to my brother who always had my back, especially during the past six years.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CSC	Criminal Smart Contract
NIZK	Non-interactive Zero Knowledge
QAP	Quadratic Arithmetic Program
SNARK	Succinct non-interactive Argument of Knowledge
UC	Universal Composability
VC	Verifiable Computation
ZK	Zero-Knowledge

Chapter 1: Introduction

Outsourcing computations to cloud services, while beneficial, raises security concerns. A remote service could manipulate the computation result, or provide approximate solutions to save resources. Therefore, in integrity-sensitive scenarios, a client will need to verify that the outsourced computation has been performed correctly. For outsourcing to be meaningful in practical settings, the cost for verifying the solution should be less than the cost of executing the computation natively. Several approaches have been proposed for solving this problem, including trusted hardware, replication and cryptography-based approaches. While the former two approaches rely on certain assumptions, the cryptography-based approaches were not considered practical until recently.

Fortunately, there has been a significant progress in the area during the past few years, which enabled efficient implementations for zk-SNARKs (Zero-Knowledge Succinct Non-interactive Arguments of Knowledge) , which allows the prover to argue for the correctness of the computation using a succinct proof that is efficient to verify [3–5]. It also enables the prover to have secret inputs to the computation. This progress has lead to developing several applications [2, 6–9], not only as academic research, but also in industry (ZCash cryptocurrency [10]).

One bottleneck in the currently used zk-SNARK constructions is the proof computation time. Such zk-SNARK constructions require representing the outsourced computation as an arithmetic circuit. As the size of the circuit (typically measured as a number of multiplications) increases, the proof computation becomes more expensive. Additionally, developing such circuits efficiently and securely is also a challenge. Most of the applications

In this thesis, we aim at addressing these challenges, by building tools and devising circuit optimizations. Additionally, motivated by the emerging cryptocurrency domain, we design and evaluate practical protocols, primarily for cryptocurrency applications, that rely primarily on verifiable computation.

1.1 Overview of Contributions

Our contributions can be categorized into two main categories: The first is developing low and high-level tools that enable programmers to efficiently develop programs for zk-SNARKs, while devising techniques for optimizing arithmetic circuit representation. The second is developing efficient verifiable computation-based protocols for cryptocurrency applications, investigating both its good and dark sides.

1.1.1 Developed Tools

In order to enable programmers and protocol designers to write efficient programs for verifiable computation, we developed the following tools:

1. xJsnark (Chapter 3): xJsnark is a high-level tool for developing efficient zk-SNARK circuits. Using previous compilers for verifiable computation like Buffet [11] or Pinocchio/Geppetto [12], programmers are assumed to have some additional experience in order to develop efficient applications on top. In addition, programmers may need to *carefully* add extra casting statements, specify additional prover inputs, or add extra constraints to the code in order to develop secure and efficient programs. xJsnark attempts to solve these problems through both the front end features, and the back end algorithms. In this work, we showed that xJsnark supports building optimized circuits for existing applications like ZeroCash [7].

   In particular, the back end of xJsnark provides new efficient, circuit-friendly algorithms for frequent operations such as memory accesses and short and long integer arithmetic; where "circuit-friendliness" in our context means that the algorithm may be expressed as a compact arithmetic circuit that minimizes the number of multiplication gates. Our new algorithms can improve the performance by constant to asymptotic factors in comparison with known approaches. The front end of xJsnark is currently developed as a Java language extension using the Jetbrains MPS framework [13]. The front end supports numerous features designed to help a non-specialist user. For example, we provide parametrized types, including bitwidth-parametrized integers, and $\mathbb{F}_p$ fields elements at the language level, allowing the user to express short and long integers very conveniently, and providing hints to the compiler for how

to optimize integer operations.

2. Low-level Tools:

   To build the protocols developed in Chapters 4 and 5, we developed low level circuit construction libraries (prior to building xJsnark). We provide more details on the lower-level tools that we developed in Appendix A.

   (a) jsnark [14]: This is a low-level circuit library written in Java, which enables the development of efficient arithmetic circuits for zk-SNARKs, on top of libsnark [5, 15], a library that implements an improved version of Pinocchio [4]. jsnark has been used for building several applications, or benchmarking [2, 8, 16, 17].

   (b) C∅C∅ [16]: This is a library for building composable zero-knowledge proofs, that are compatible with the Universal Composability framework [18]. To realize such proofs efficiently, this work also explores selected known cryptographic primitives such as encryption and key exchange, and expresses them as efficient arithmetic circuits (Details are in Appendices A and B). This was part of a joint work with Zhichao Zhao, Andrew Miller, Yi Qian, Hubert Chan, Charalampos Papamanthou, Rafael Pass, abhi shelat and Elaine Shi.

1.1.2 Protocols and Applications

In this direction, we explore how verifiable computation can be used to build practical protocols for cryptocurrency applications. Motivated by the emerging

smart contract technology, we make the following two contributions:

1. HAWK [2]: A decentralized system for privacy-preserving smart contracts. Existing smart contract systems, like Ethereum [19] lack transactional privacy, as for transparency purposes, miners have access to the plaintext values of the inputs. HAWK utilizes the new breakthroughs in verifiable computation techniques to solve this problem, and enables applications like decentralized private auctions and crowdfunding to run on a public blockchain system. A HAWK contract relies on a special party called the manager, that is only trusted for the privacy of the inputs, which are only revealed after all parties commit to them. The manager is not trusted for the correctness of the computation. Even if the manager deviates from the protocol or colludes with one of the participants, this will not affect the outcome of the execution. Chapter 4 illustrates the design and implementation of HAWK. This work appeared previously in IEEE S&P 2016 as a joint work with Andrew Miller, Elaine Shi, Zikai Wen and Charalampos Papamanthou.

2. How smart contracts can be used for crime [8]: The introduction of new technologies usually enables new forms of crimes. For example, technologies like the internet, social networks and Bitcoin lead previously to new types of crimes or facilitated the existing ones. Therefore, from a proactive perspective, we investigate how smart contracts and progress in verifiable computation schemes could facilitate serious crimes in the future. One of the crimes we investigated was cryptographic key theft which relied mainly on a practical verifiable com-

putation protocol that could allow criminals to use a smart contract system to fairly exchange such information. We discuss this work in Chapter 5. This work has appeared in CCS 2016, and is a joint work with Ari Juels and Elaine Shi.

Additional future directions and open problems are discussed in Chapter 6.

Chapter 2: Background and Preliminaries

In this chapter, we provide background about verifiable computation and smart contracts.

2.1 Verifiable Computation

A verifiable computation (VC) scheme involves two parties, a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, where $\mathcal{P}$ proves the correctness of executing a program $\mathcal{F}$ on an input x from $\mathcal{V}$, and (optionally) a secret input u from $\mathcal{P}$. $\mathcal{P}$ sends $\mathcal{V}$ both the output $\vec{y}$ and a proof $\vec{\pi}$ to verify the result. Specifically, a VC scheme typically consists of three algorithms [20]:

- $(PK_{\mathcal{F}}, VK_{\mathcal{F}}) \leftarrow \mathsf{KeyGen}(\mathcal{F}, 1^{\lambda})$ A randomized algorithm that given an outsourced program $\mathcal{F}$ and a security parameter λ, it outputs a public proving key $PK_{\mathcal{F}}$ and a verification key $VK_{\mathcal{F}}$. The verification key might be public or private depending on the setting.

- $(y, \pi) \leftarrow \mathsf{Prove}(\mathcal{F}, PK_{\mathcal{F}}, x, u)$. The prove algorithm for a program $\mathcal{F}$ takes the public proving key $PK_{\mathcal{F}}$, the public input $\vec{x}$, the prover's secret input u, and outputs $\vec{y} \leftarrow \mathcal{F}(x, u)$, and the proof π proving the correctness of the computation.

- $\{0,1\} \leftarrow \mathsf{Verify}(VK_{\mathcal{F}}, x, y, \pi)$. Given the verification key $VK_{\mathcal{F}}$, the proof $\vec{\pi}$, and the statement (x, y), the Verify algorithm outputs 1 if $y = \mathcal{F}(x, u)$.

In many practical verifiable computation settings, it is desirable to reduce the size of the proofs and the effort of the verifier with respect to the computation being verified. Thankfully, recent research progress have shown how to construct practical VC schemes using a cryptographic primitive called: zero-knowledge Succinct Non-interactive ARgument of Knowledge (zk-SNARKs). Most notably, existing implementations for zk-SNARKs have enabled to have constant size proofs and minimal verification time under certain assumptions [3, 4].

For simplicity, we mainly focus on what is called preprocessing zk-SNARKs, in which the whole program is represented as a single circuit or a single system of constraints, and a one-time trusted setup per each different circuit is needed in the beginning. However, the techniques we describe for circuit optimizations can also be extended in a straightforward way to systems that support recursive composition of zk-SNARKs [12, 21]. The formal security definitions for (zk-)SNARKs can be found in Section 2.3.

2.1.1 Program Representation and Cost model

In the preprocessing zk-SNARKs we are considering, the program to be verified is expressed as a Quadratic Arithmetic Program (QAP) [3, 20], where computations are represented as a set of quadratic equations over a finite field (typically a 254-bit prime field p), or in other words, a circuit of additions and multiplications gates mod

p. We denote each quadratic equation or a multiplication operation as a *constraint*.

To translate a program into set of constraints, the main operations can be translated as follows:

Arithmetic Operations (mod p). Translation of addition and multiplication (mod p) is straightforward. Note that additions and multiplication by constants are almost free operations, while each multiplication gate costs one constraint.

Bitwise Operations. Access to the individual bits of a wire in the circuit is an expensive operation. For example, for an n-bit wire w, it would require $n + 1$ constraints to verify that each bit wire b_i achieves the following constraint: $b_i(1 - b_i) = 0$, and that all bits achieve the constraint $(\sum 2^i b_i) \times 1 = w$. We denote this gate as *split* gate, following the naming of Pinocchio, while the reverse operation as *pack* gate following the naming convention of libsnark. Note that the pack operation can be just implemented as a weighted linear combination of the bits. For any two bits b_i, and b_j, the operations: AND, OR, and XOR each costs one constraints, while the negation of a certain bit can be implemented as a linear combination of the bit.

Arithmetic Operations (mod $p' \neq p$). The problem of representing arithmetic operations is more challenging when the modulus is not equal to p. For example, when the operations are done over $p\prime = 2^n$, to obtain a correct result, a remainder operation needs to be applied on the result, which leads to a number of constraints that are at least equal to the bitwidth of the result, as it requires at least one split gate. However, as we see in Chapter 3, the compiler can apply some heuristics for efficient translations of such operations. Previous compilers such as [11, 12, 20]

assume that the programmer is responsible for taking such decisions.

Conditional Statements and Control Flow. In many programs, various kinds of checks are needed to control the program flow and computation results. This for example includes equality checks, comparison checks and results of Boolean statements. Checking the equality of two n-bit wires requires two constraints (assuming $n < 254$), while the unsigned integer comparison costs about $n + 2$ constraints.

Based on the conditional results, the flow of the program may could be decided, but since all the computation is modeled as a circuit, all the possible execution paths are encoded in advance in the circuit, but only the valid execution path should have an impact on the result. Efficient methods for realizing such encoding exist. For example, the Fairplay compiler [22] provides an efficient approach for encoding the assignment operations in different execution paths. This approach is inherited by Buffet [11], since it is built based on the Fairplay compiler front end. Geppetto [12] on the other hand applies an orthogonal optimization to reduce the effect of the paths that were not taken on the proof time.

Assertions (mod p). This refers to gates that verify a constraint given input wires. In its general form, an assertion gate accepts three inputs a, b and c, and verifies that $a \times b = c$.

Memory/Array Access. Typically, it's straightforward to represent the array access operations, when the indices are known during compilation time. The problem is challenging when the access index is not known in advance. A solution using the Pinocchio compiler [20] will perform a linear scan over an n element array, costing $3n$ constraints.The Pepper compiler [23] provides a Merkle tree approach using an

Ajtai hash function, costing about $4000 \log n$ constraints. Due to the large multiplicative constant in the Merkle tree approach, a method for verifying memory access using permutation networks were proposed in [24], and subsequently used in TinyRAM [25], and Buffet [11].

It should be noted that what makes the problem of optimized program representation for preprocessing zk-SNARKs interesting is that its cost model is different from the cost models of usual programs. In Chapter 3, we will show how this can be used to develop new techniques.

2.1.2 Existing Development Tools

The currently used utilities for developing verifiable programs span two different categories:

High-level language compilers. This includes many works such as Pinocchio [20], TinyRam [25], Pantry [23], Buffet [11], and Geppetto [12]. Pinocchio's compiler translates a subset of the C programming language to an arithmetic file that provides a circuit representation of the computation to be verified. Plus its support for large-scale computations via Multi-QAPs, Geppetto's compiler provides additional features over Pinocchio's compiler, e.g. enabling programmers to define long integer types, specify bounding constraints in the code and access to bit values. Geppetto's compiler also employs energy-saving circuits, by which the prover's cost gets minimized in branches that are not taken during execution. Pantry and Buffet support a larger subset of the C language. Pantry was the first to extend verifiability to

computations with state, such as map-reduce jobs. Buffet provides more efficient control flow, and random memory accesses, combining the permutation network approach with compiler optimizations. TinyRam compiles high-level C programs to TinyRam assembly instructions, and uses a universal circuit that does not require a set up each time, but this results in higher cost per program step.

Although the above tools include many theoretical and engineering optimizations, it is not straightforward to develop programs efficiently, especially for cryptographic operations.

Low-level circuit construction tools. Although such tools require more programming effort, they were used in many applications that require optimized performance [2, 7, 8]. This for example includes libsnark's gadget libraries [15]. In libsnark's C++ libraries, a programmer represents the verifiable program as gadgets connected together. Each gadget defines a set of constraints, and how to set the value of its output variables. jsnark provides a simpler Java interface to libsnark so that it can make development easier and likely to produce more efficient circuits, and it uses the same libsnark cryptographic back end eventually. Other works include snarklib [26] , and bellman [27].

2.2 Smart Contracts

In this section, we cover some basics about smart contracts which are needed in Chapters 4 and 5.

New cryptocurrencies [19,28] rely on the novel blockchain technology where the

network reaches consensus not only about data, but also computations performed on this data. In Bitcoin for example, miners verify transactions and store valid ones in a global ledger. As long as the majority of the network is honest, the consistency of the ledger is maintained. However, Bitcoin supports a limited scripting language which limits the types of functions that can run on top of the blockchain. Although there were previous attempts at building smart contracts on top of the Bitcoin blockchain [29–33], building new custom applications is still not straightforward.

Fortunately, new decentralized cryptocurrency systems such as the recently launched Ethereum [19] enable a richer form of smart contracts that allow more general computations. Smart contracts in such systems act as autonomously executing trusted parties, who control the flow of money based on a predefined set of rules that cannot be changed by anyone in the network (assuming honest majority). This motivated many decentralized applications to be built, like prediction markets, crowdfunding applications and others [34–37].

Figure 2.1 shows the high-level architecture of a smart contract system instantiated over a decentralized cryptocurrency such as Bitcoin or Ethereum. When the underlying consensus protocol employed the cryptocurrency is secure, a majority of the miners (as measured by computational resources) are assumed to correctly execute the contract's programmable logic.

Cost model To protect decentralized smart contract systems against denial of service and infinite loop attacks, smart contracts in Ethereum has the notion of *gas*, which is proportional to the amount of computation and storage that a miner

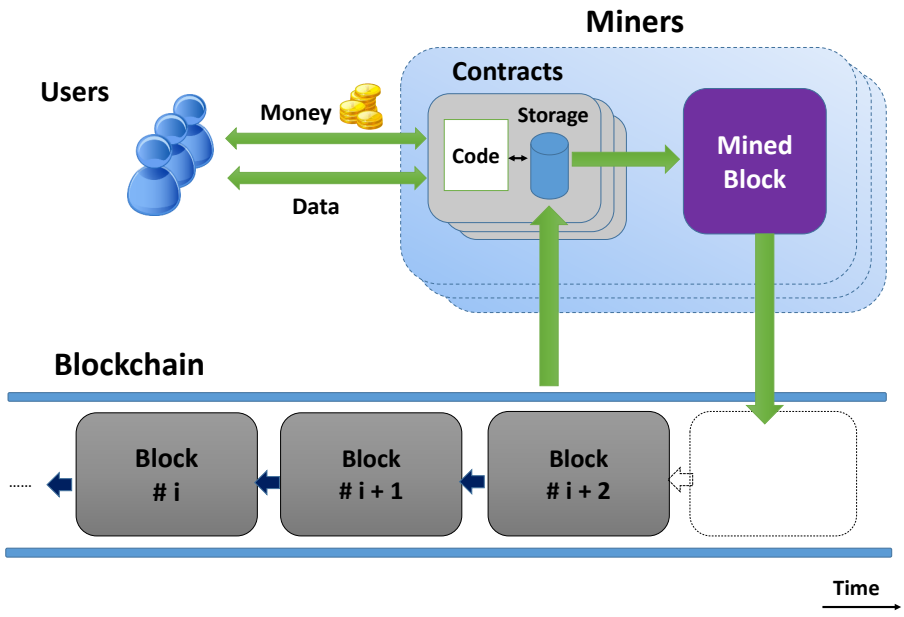

Figure 2.1: **Schematic of a decentralized cryptocurrency system with smart contracts**. A smart contract's state is stored on a public blockchain system. A smart contract program is executed by a network of miners who reach consensus on the outcome of the execution, and update the contract's state on the blockchain accordingly. Users can send money or data to a contract; or receive money or data from a contract.

consumes while executing a contract functionality. Users of smart contracts have to pay for gas consumed by contracts execution.

In the contracts developed in this thesis (Chapters 4 and 5), the gas is not explicitly expressed in the notation. However, one of our objectives is to minimize the computation that needs to be performed by the miners, while maintaining privacy. Therefore, we rely on zk-SNARKs as a main tool, as it enables succinct proofs and efficient verification compared to other available zero knowledge proof constructions.

2.2.1 Limitations of smart contracts

Smart contracts enable many beneficial use cases, including financial instruments, decentralized auctions and crowdfunding, .. etc. As in the case of Bitcoin, no entity can manipulate the execution of a contact, and hence smart contracts remove the need for trusted intermediaries or reputation-based systems. In comparison with Bitcoin, smart contracts enable fair exchange between distrustful parties based on a programmable logic. This feature can enable protection against cheating or aborting adversaries in complex protocols.

However, from another perspective, existing smart contract systems have a couple of drawbacks which we address or explore in Chapters 4 and 5.

First, they lack transactional privacy. In order to perform computations in an efficient trustworthy manner on top of a blockchain, the contract logic has to have access to all the input values of the computation. Additionally, the result of contract execution is also public. This may not fit many applications where privacy is essential. Therefore, in this thesis (Chapter 4), we show how we can develop privacy-preserving smart contracts that aim to solve this problem.

Second, as Bitcoin stimulated a lot of criminal activity due to its pseudonymity, we expect that smart contracts could also stimulate further criminal activity, primarily due to their richer functionality. For example, they can enable complex fair exchange between mutually distrustful criminal parties, eliminating the need for reputation systems or third-party intermediaries that could be manipulated or infiltrated by law enforcement [38, 39]. In Chapter 5, we illustrate how smart contracts

could be used for criminal activities, to advise the community against their risks.

2.3 Cryptographic Preliminaries

In this section, we review the formal definitions for zero knowledge proofs and zk-SNARKs [16].

Notation. In the following, $f(\lambda) \approx g(\lambda)$ means that there exists a negligible function $\nu(\lambda)$ such that $|f(\lambda) - g(\lambda)| < \nu(\lambda)$.

2.3.1 Non-Interactive Zero-Knowledge Proofs

A non-interactive zero-knowledge proof system (NIZK) for an NP language $\mathcal{L}$ consists of the following algorithms:

- $\mathsf{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L})$, also written as $\mathsf{crs} \leftarrow \mathsf{KeyGen}_{\text{nizk}}(1^\lambda, \mathcal{L})$: Takes in a security parameter λ, a description of the language $\mathcal{L}$, and generates a common reference string crs.

- $\pi \leftarrow \mathcal{P}(\mathsf{crs}, \mathsf{stmt}, w)$: Takes in crs, a statement stmt, a witness w such that $(\mathsf{stmt}, w) \in \mathcal{L}$, and produces a proof π.

- $b \leftarrow \mathcal{V}(\mathsf{crs}, \mathsf{stmt}, \pi)$: Takes in a crs, a statement stmt, and a proof π, and outputs 0 or 1, denoting accept or reject.

- $(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \leftarrow \widehat{\mathcal{K}}(1^\lambda, \mathcal{L})$: Generates a simulated common reference string $\widehat{\mathsf{crs}}$, trapdoor τ, and extraction key ek

- $\pi \leftarrow \widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \mathsf{stmt})$: Uses trapdoor τ to produce a proof π without needing a witness

Perfect completeness. A NIZK system is said to be perfectly complete, if an honest prover with a valid witness can always convince an honest verifier. More formally, for any $(\mathsf{stmt}, w) \in R$, we have that

$$\Pr\left[\begin{array}{l} \mathsf{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}),\ \pi \leftarrow \mathcal{P}(\mathsf{crs}, \mathsf{stmt}, w) : \\ \quad \mathcal{V}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1 \end{array}\right] = 1$$

Computational zero-knowledge. Informally, a NIZK system is computationally zero-knowledge if the proof does not reveal any information about the witness to any polynomial-time adversary. More formally, a NIZK system is said to computationally zero-knowledge, if there exists a polynomial-time simulator $S = (\widehat{\mathcal{K}}, \widehat{\mathcal{P}})$, such that for all non-uniform polynomial-time adversary $\mathcal{A}$,

$$\begin{aligned} & \Pr\left[\mathsf{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}) : \mathcal{A}^{\mathcal{P}(\mathsf{crs}, \cdot, \cdot)}(\mathsf{crs}) = 1\right] \\ \approx\ & \Pr\left[(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \leftarrow \widehat{\mathcal{K}}(1^\lambda, \mathcal{L}) : \mathcal{A}^{\widehat{\mathcal{P}}_1(\widehat{\mathsf{crs}}, \tau, \cdot, \cdot)}(\widehat{\mathsf{crs}}) = 1\right] \end{aligned}$$

In the above, $\widehat{\mathcal{P}}_1(\widehat{\mathsf{crs}}, \tau, \mathsf{stmt}, w)$ verifies that $(\mathsf{stmt}, w) \in \mathcal{L}$, and if so, outputs $\widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \mathsf{stmt})$ which simulates a proof without knowing a witness. Otherwise, if $(\mathsf{stmt}, w) \notin \mathcal{L}$, the experiment aborts. This notion is *adaptive* zero knowledge in the sense that the simulator must specify the reference string before seeing the theorem statements.

Computational soundness. A NIZK scheme for the language $\mathcal{L}$ is said to be

computationally sound, if for all polynomial-time adversaries $\mathcal{A}$,

$$\Pr\left[\begin{array}{l}\mathsf{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}), (\mathsf{stmt}, \pi) \leftarrow \mathcal{A}(\mathsf{crs}) : \\ \\ (\mathcal{V}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1) \wedge (\mathsf{stmt} \notin \mathcal{L})\end{array}\right] \approx 0$$

Simulation extractability. Simulation extractability is a strong notion which requires that even after seeing many simulated proofs (even for false theorems), whenever the adversary makes a new proof, a simulator is able to extract a witness. Simulation extractability implies simulation soundness and non-malleability (i.e., it is not feasible for an adversary to take a verifying proof and "maul" it into a verifying proof for another statement) since if the simulator can extract a valid witness from an adversary's proof, the statement must belong to the language. More formally, a NIZK system is said to be simulation extractable if it satisfies computational zero-knowledge and additionally, there exists a polynomial-time algorithm $\mathcal{E}$, such that for any polynomial-time adversary $\mathcal{A}$, it holds that

$$\Pr\left[\begin{array}{l}(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \leftarrow \widehat{\mathcal{K}}(1^\lambda, \mathcal{L}); \\ \\ (\mathsf{stmt}, \pi) \leftarrow \mathcal{A}^{\widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \cdot)}(\widehat{\mathsf{crs}}, \mathsf{ek}); \\ \\ w \leftarrow \mathcal{E}(\widehat{\mathsf{crs}}, \mathsf{ek}, \mathsf{stmt}, \pi) : \mathsf{stmt} \notin Q \text{ and} \\ \\ \qquad\qquad (\mathsf{stmt}, w) \notin \mathcal{L} \text{ and } V(\widehat{\mathsf{crs}}, \mathsf{stmt}, \pi) = 1\end{array}\right] \approx 0$$

where in the above, Q is the list of oracle queries made by $\mathcal{A}$ to $\widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \cdot)$. Here the $\widehat{\mathcal{K}}$ is identical to the zero-knowledge simulation setup algorithm.

2.3.2 zk-SNARKs

A zk-SNARK is a NIZK that is succinct, perfectly complete, computationally zero-knowledge, and has a knowledge extractor:

Succinctness. A SNARK is said to be succinct if an honestly generated proof has $\mathsf{poly}(\lambda)$ bits and that the verification algorithm $\mathcal{V}(\mathsf{crs}, \mathsf{stmt}, \pi)$ runs in $O(|\mathsf{stmt}| \cdot \mathsf{poly}(\lambda))$.

Adaptive knowledge extraction. Knowledge extraction requires that if a proof generated by an adversary is accepted by the verifier, then the adversary "knows" a witness for the given instance; i.e., there exists an algorithm $\mathcal{E}$ which recovers a witness. Furthermore, the extraction property holds adaptively even if the prover picks the statement after seeing the reference string. Formally, a SNARK for language $\mathcal{L}$ satisfies the knowledge extraction property *iff:*

For all polynomial-sized adversary $\mathcal{A}$, there exists a polynomial-size extractor $\mathcal{E}_{\mathcal{A}}$, such that for all advice strings $z \in \{0,1\}^{\mathsf{poly}(\lambda)}$,

$$\Pr\left[\begin{array}{l|ll} & & \mathcal{V}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1 \\ \mathsf{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}) & & \\ (\mathsf{stmt}, \pi) \leftarrow \mathcal{A}(\mathsf{crs}, z) & : & \\ & & (\mathsf{stmt}, a) \notin \mathbb{R}_{\mathcal{L}} \\ a \leftarrow \mathcal{E}_{\mathcal{A}}(\mathsf{crs}, z) & & \end{array}\right] \approx 0$$

Chapter 3: xJsnark: A Framework for Efficient Verifiable Computation

3.1 Overview

Existing zk-SNARK constructions model computation as arithmetic circuits (or alternatively, as a set of arithmetic constraints) over a finite field. As mentioned previously, there are two methods for a programmer to express the computation that needs to be verified, either through compilation from a high-level language [4,11,12], or by manual circuit construction frameworks [5,14]. The former provides programmers convenience; while the latter enables lower-level control and optimizations, resulting in possibly much better performance than circuits auto-generated by a compiler, but requires effort and knowledge from the programmer.

In this work, our goal is to bridge this gap. We design and implement xJsnark, a programming framework for developing (zk-)SNARK applications. xJsnark takes a language-compiler co-design approach: It introduces user- and compiler-friendly language features that allow the user to conveniently write programs in a Java-like language and subsequently enable the back end to extract additional information needed for converting the user-supplied program to a compact, optimized circuit.

As we show later, xJsnark reduces programmer effort in comparison with existing SNARK compilers such as Buffet [11] and Geppetto [12]; and meanwhile improves the performance of the compiled SNARK implementation by **1.2**× to more than **3**× for different cryptographic and memory access applications. We will also illustrate how the framework reduces the effort in developing large circuits as in the case of ZeroCash [7], while producing optimized output.

3.1.1 Problem Statement

An important goal of xJsnark is "program-to-circuit" conversion, i.e., to compile a user-supplied program described in a Java-like source language into a compact circuit representation that is recognized by existing SNARK schemes. At the moment, xJsnark emits circuits in a libsnark-compatible format [15], such that the resulting SNARK can be executed using the libsnark back end. Thus our contribution is not the back end SNARK implementation, but rather, the program-to-circuit conversion stage, and the co-design of the source language and compile-time optimizations to minimize the compiled circuit.

This problem of program-to-circuit conversion is commonly encountered in designing programming frameworks for cryptography: besides (zk-)SNARKs, it has also been investigated in the context of secure multi-party computation [22,40–42] — in particular, known cryptographic building blocks for securing the integrity and/or confidentiality of computation customarily express computation as circuits.

SNARK-specific program-to-circuit conversion. Several factors make the

program-to-circuit conversion problem unique in the SNARK context, and our algorithmic techniques described later would repeatedly make use of these optimizations to achieve constant to asymptotic performance improvements over existing approaches.

First, an important observation that fundamentally differentiates circuit generation in the SNARK context than, say, in the multi-party computation context [22, 40–43], is the following: a SNARK circuit need not necessarily compute a function in the forward direction, it suffices to generate a circuit that verifies the correct of the computation result — and the latter is often much cheaper than the former. For example, the statement $y = x/a$ can be verified much more efficiently by checking that $y \times a = x$ rather than computing the division in the forward direction. This observation has also been pointed out by several earlier works [4, 6, 11, 12, 16, 24, 25] — but in this work we will apply it in new ways in the design of several circuit-friendly algorithms that achieve constant to asymptotic performance improvements over existing approaches.

Second, known SNARK constructions rely on arithmetic circuits over a finite field. Moreover, known SNARK implementations have a unique performance profile where multiplication of two variables are expensive; whereas addition gates or multiplication with predetermined constants come almost for free (See Section 2.1.1). Therefore, the optimization metrics are very different from conventional compilers in our case. We focus on how to emit arithmetic circuits that express a user-supplied program while minimizing the number of expensive multiplication gates.

3.1.2 Technical Highlights

To emit compact circuits for user-supplied programs, we introduce new algorithmic techniques in all stages of the compilation. Our new algorithms *improve the circuit size by constant to asymptotic factors* for frequent building blocks relative to the state-of-the-art, while not requiring as much experience from the programmer compared to earlier compilers.

New circuit-friendly building blocks. First, at the building block level, we design new efficient, circuit-friendly algorithms for frequent operations such as memory accesses and short and long integer arithmetic; where "circuit-friendliness" in our context means that the algorithm may be expressed as a compact arithmetic circuit that minimizes the number of multiplication gates (more specifically, multiplication of two variables and not with a predetermined constant). Our new algorithms can improve the performance by constant to asymptotic factors in comparison with known approaches. More specifically, we make the following algorithmic contributions:

- *Efficient read-only memory.* We present an algorithm (Section 3.4) for verifying a batch of k read-only memory accesses in total cost proportional to $k\sqrt{n}$ where n is the size of the memory array to be accessed, where cost is expressed in terms of the number of arithmetic multiplications of two variables — note that the number of addition gates and "multiply by constant" gates are still linear in n, but as mentioned earlier these gates come almost for "free". (Note

that earlier work on Boolean circuits obtained similar bounds for the multiplicative complexity of a Boolean function [44]. In contrast, in this work we consider the case of arithmetic circuits, while relying on the observation that multiplications by constants are for free, and on external witnesses to provide efficient checks). We will show that for a broad range of choices over k and n, our read-only memory access algorithm outperforms the state-of-the-art by factors ranging between **3-10**×, and overall we illustrate how it can improve the AES implementation by more than 2×.

- *Smart memory.* Our xJsnark framework supports a smart memory algorithm that adapts the memory implementation to obtain high efficiency. Depending on the concrete value of k and n, and whether the memory access is read-only, our back end automatically selects the most efficient memory access algorithm among the following: 1) the naïve linear sweep algorithm which may be efficient for sufficiently small values of k; 2) a permutation network [11, 24, 25], and 3) our new read-only memory algorithm mentioned above.

- *Long integer arithmetic.* We introduce several new circuit-friendly algorithms for efficient long integer arithmetic. xJsnark internally expresses long integers as an array of short integers whose bitwidth can fit in the SNARK's native arithmetic field. Henceforth let m denote the length of this array.

    1. *Multiplication.* Known works employ either a naïve multiplication algorithm that incurs $\Theta(m^2)$ circuit size (e.g., Cinderella [9]); or adopt Karatsuba [45] that incurs $\Theta(m^{1.58})$ circuit size (e.g., ObliVM [40], GraphSC [43])

— where circuit size counts only multiplication gates. We propose a SNARK-friendly long-integer multiplication algorithm that incurs only $O(m)$ multiplication (by non-constant) gates.

2. *Modular arithmetic.* Modular arithmetic is a frequently encountered operation for implementing a wide class of cryptographic algorithms (e.g., RSA circuits) in SNARKs. To support modular arithmetic, a recurring operation is to verify the modular congruence of two variables, i.e., verify that $a \equiv b * q + r$ where q is the modulus.

   Using our long integer multiplication technique as a building block, we devise an improved algorithm for checking the modular congruence of long integers, leading to an improvement of **3**× for this operation, and improving the overall performance by more than **1.5**× relative to the state-of-the-art [9] that was built on top of Geppetto [12]. This is while minimizing the programmer's effort/experience requirements.

Global optimizations for integer arithmetic. Besides optimizing individual building blocks, our compiler also makes (somewhat) global optimization decisions for frequent operations such as integer arithmetic. One challenge in supporting bitwidth-parametrized integers is to figure out when to perform bitwidth realignment. More specifically, imagine that the program contains operations on `uint32` variables, i.e., unsigned integers of 32 bits. Since the SNARK's native field is much larger than 32 bits, we need not perform a $\mod 2^{32}$ operation for each arithmetic operation (henceforth referred to as bitwidth realignment). One naïve strategy (i.e.,

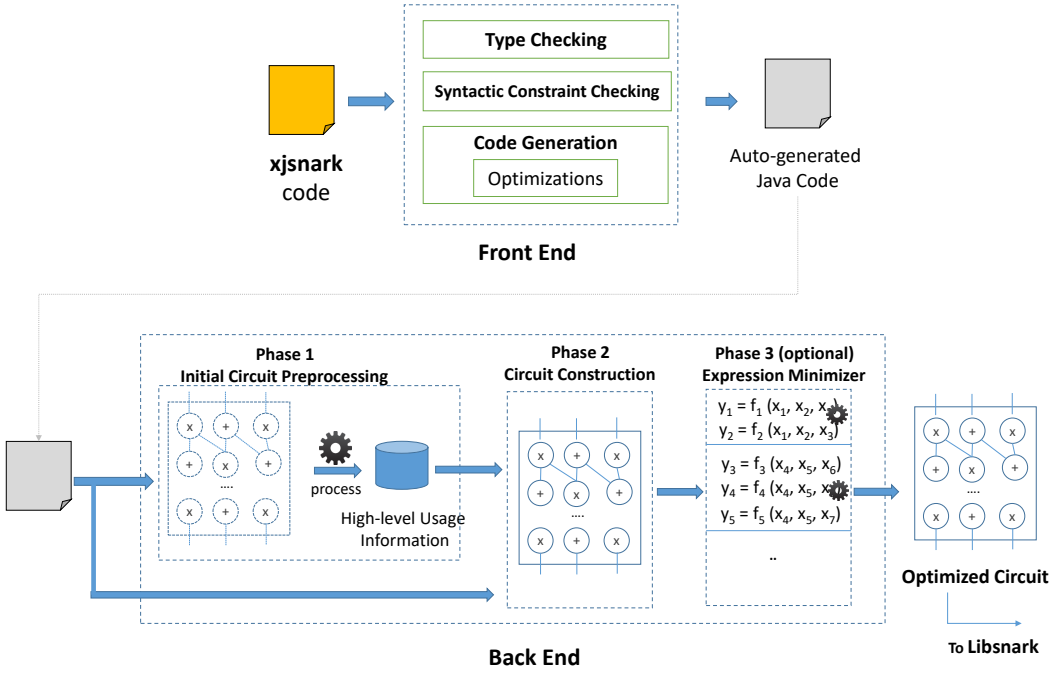

Figure 3.1: xJsnark Overview

the lazy strategy) is simply let the bitwidth grow but keep track of the maximum bitwidth of internal variables — then we only perform realignment whenever an overflow is just about to happen. As we show later, this naïve lazy strategy is not the optimal. Instead, our compiler is able to perform more globally aware decisions as to when to perform bitwidth realignment.

Circuit minimization. Third, we implement a customized version of the state-of-the-art circuit minimization techniques — more specifically, multi-variate polynomial minimization techniques — to minimize the generated circuit. Such circuit minimization techniques may have exponential time, and therefore we devise algorithms to cluster the arithmetic constraints to be verified into bounded-size groups, and we apply multi-variate polynomial minimization to each group.

3.1.3 Implementation, Evaluation, and Open Source

Besides our new algorithmic techniques and various more globally aware optimizations, one important contribution we make is to integrate all these techniques into a unified, user-friendly programming framework. We hope that the xJsnark user can benefit from our effort and be able to develop efficient SNARK implementations without needing much specialized knowledge on the topic. To this end, we released our xJsnark framework as an open source project [46].

Implementation. We present an overview of our xJsnark framework in Figure 3.1. xJsnark's front-end is developed atop Java using Jetbrains MPS, an open-source project [13] for implementing domain-specific languages. xJsnark's compilation back-end encompasses several stages:

- First pass: the back-end collects useful information about the structure of the circuit, e.g., how variables are being used (e.g. whether involved in arithmetic or Boolean operations, and how many times used), how memory is being accessed, etc. This is done by creating a dummy circuit that does not realize every low-level detail, but only what is needed to understand the characteristics of the circuit.

- Second pass: making use of information collected during the first pass, the back end decides an efficient circuit representation of the computation.

- Optional third pass: This pass uses a customized multivariate polynomial minimization technique to introduce more savings in the circuit.

Front end. The front end of xJsnark is developed as a Java language extension using the Jetbrains MPS framework [13]. xJsnark's front end supports numerous features designed to help a non-specialist user. First, we provide parametrized types, including bitwidth-parametrized integers, and $\mathbb{F}_p$ fields elements at the language level, allowing the user to express short and long integers very conveniently. The extension comes with an Interactive Development Environment (IDE) that is based on projectional editing and real-time type checking that allows programmers to detect programming errors early on. We provide code examples in Appendix C.1, and discuss the trade-offs of using the underlying framework in Section 3.2.1.

Using previous compilers like Buffet [11] or Geppetto [12], programmers are assumed to have some additional experience in order to develop efficient applications on top. For example, programmers may need to *carefully* add extra casting statements, specify additional prover inputs or add extra constraints to the code in order to develop secure and efficient programs. xJsnark attempts to solve these problems through both the front end features, and the back end algorithms.

Performance. For four different cryptographic applications spanning SHA-256, SWIFFT hash function, RSA, and AES, we illustrate how xJsnark can produce more efficient circuits by factors ranging between 1.2× to more than 2×, while not requiring the programmer to be experienced in the underlying SNARK implementation (Section 3.6.1). Additionally, we show how our framework produces efficient random access circuits by a factor of 2-3× in the case of sorting, while also provid-

ing more efficient ways to obtain more concise circuits that are order of magnitudes better (See Section 3.6.2). Furthermore, we illustrate that the framework can produce efficient circuits as done by existing low-level implementations, as in the case of ZeroCash [7] when developed in our framework (Section 3.6.3).

3.2 xJsnark's Front End

In this section, we discuss the language extension features. Using our Java extension built on top of Java using Jetbrains MPS, a programmer will specify the code for the computation to be verified. Code examples are provided in Appendix C.1. First, we provide brief background and discussion of Jetbrains MPS in Section 3.2.1, then we discuss the front end features.

3.2.1 Language Development using MPS

Our language extension for the front end is built using Jetbrains MPS [13], an open source language workbench [47] based on projectional editing. In this section, we provide some background on language workbenches and Jetbrains MPS.

Language workbenches have been developed to facilitate the development of new general-purpose or domain-specific languages. They can be generally classified into parser/text-based development tools, such as Xtext [48], and projectional editor-based workbenches such as Jetbrains MPS that we use here [49]. Projectional editing is a technique that allows the programmers to manipulate the abstract syntax tree of a program directly, without relying on parsers/grammars.

Jetbrains MPS provides flexibility in defining new language extensions, and in modular composition of languages. The MPS approach has been used already to develop different domain-specific languages, including *mbeddr* [50] which provides extensions on top of the C language for embedded system development. Other examples include Youtrack [51], an issue tracking system that has a Java language extension for working with persistent data and queries among others [52], and MetaR which facilitates biological data analysis with the R language [53]. For the drawbacks, the use of projectional editing is less common than text editing for general programming, however the usage of a projectional editor enables the modular composition of language extensions in a more flexible way. Additionally, Jetbrains MPS attempts to handle most of the usability issues that arise from projectional editing [54], while the mbeddr authors [55] argue that their pilot usability study suggest a quick learning curve for the end language users to get familiar with the editor. In the discussion section (Section 3.7), we discuss other future plans to investigate the usage of other front ends for our optimizations and algorithms in the back end.

In the following, we give a brief idea about the necessary elements that a Java language extension on top of MPS Jetbrains needs to have [56]. To define a language extension, such as the one that we use in this work, the following modules need to be defined. Some details are omitted/simplified for brevity.

Abstract syntax: The first step is to define the *structure*, by specifying the additional abstract syntax tree nodes required for the extension. This is done through the definition of new *concepts*. A concept definition typically includes the properties, children and references of each node. Then, the *constraints* on the structure

are defined, to specify any restriction on the properties, children and references of any concept.

Editor (Concrete Syntax): This specifies the projectional editor behavior for the new concepts, e.g. the visualization of the newly added extensions, and the automated actions by the editor.

Type system: This specifies the type system rules needed for any introduced new types.

Code generation: This specifies how the extension constructs will be translated to the base Java language, based on the definition of reduction rules.

After the language developer specifies the above, the user of the extended language will be able to write programs in the new language with IDE support, e.g. auto-completion, error highlighting, and others.

3.2.2 Extension Features

To support easier development of arithmetic circuits, while giving optimization hints to the back end, xJsnark's front end supports the following:

3.2.2.1 Parametrized Types

To give the programmer greater control, and in the same time enable our back end implementation to translate the code efficiently, our framework introduces parameterized types for integers and field elements, where the programmer can specify the bitwidth of integers, and the modulus of the field elements. The following

snippet shows examples of variables declared using those types.

```
uint_7 x1 = 12;
uint_1024 x2 = 810527815761576416536152365131611232 3u;
F_swifft y1 = 123;
F_p256 y2 = F_p256(810527815761576416536152365131u);
bit b1 = 1;
```

Note that the programmer easily specifies long integer and field element types, without specifying how that will be implemented in the background. Also, the programmer can specify long integer literals without dividing them to chunks according to the native underlying field. In order to enable the programmer to define finite field types, the framework has a special file where field identifiers can be specified, typically in the following syntax:

```
swifft : 257
p256: .. // NIST Curve P−256 prime
```

Then, the programmer can use these identifiers when defining field elements.

3.2.2.2 Operators

The framework allows the programmer to directly use typical arithmetic operators with the types defined above (e.g. $+, -, *, /, \&, |,\ $) when applicable, instead of using special methods. For example, the following snippet shows a piece of code that verifies the ownership of an RSA secret key:

```
Program RSA_SecretKey_Knowledge{
  uint_2048 modulus;
  uint_1024 p;
```

```
  uint_1024 q;

  input{ modulus };
  output { };
  witness {p, q};

  void Main(){
    verifyEq((uint_2048)p*q, modulus); // Equality Assertion
  }
}
```

Due to the underlying Java implementation in MPS Jetbrains, we introduce new operators for bit and equality operators such as (AND, OR, NOT) to be compatible with our types. We also introduce new operators like `inv`, which obtains the multiplicative inverse of a field element (assuming a prime order).

3.2.2.3 External Code Blocks

In many cases, computing the value of a prover's witness can be more expensive than its verification, e.g. verifying a solution for a linear system of equations has less complexity than computing the solution itself. In such cases, previous compilers assume that the computation of such witnesses happens independently outside the circuit, and only the verification is specified in the code. We believe this may be inconvenient, due to writing code in two different frameworks. Instead, the programmer can specify in our framework within the same code how these computations will take place in Java. To specify code to be executed outside the circuit, the xJsnark programmer can use the `external` code blocks, and the `val` operator, which refers to the value during runtime (See Appendix C.1.1).

3.2.2.4 Smart Memory and Permutation Verifier

As we will discuss in detail in the back end, xJsnark provides a smart memory implementation that decides the best way to translate memory operations after analyzing the workload of each array. Additionally, xJsnark provides a native function that can be used to verify that a group of elements is a permutation of another, without exposing the programmer to the internal details of switching networks. This feature can be used along with the external code block and constraints to compile some applications more efficiently, e.g. sorting with respect to arbitrary criteria (See Appendix C.1.1 for an example, and Section 3.6.2 for performance results).

3.2.2.5 Composite Types

xJsnark allows the programmer to define class types that are a collection of xJsnark's primitive types. It also supports a very basic notion of object-oriented programming, such that these classes could also define methods that manipulate the state of the object. These classes have a special implementation in the back end, so that the programmer can easily manipulate them as circuit inputs/outputs/witnesses. The programmer can use the keyword `struct` to define such classes.

3.2.2.6 Assertions

xJsnark provides supports for writing high-level constraints in the code. As in the previous example, the keyword `verifyEq` forces two values to be exactly the same. Additional assertion keywords include `verify` to verify general constraints

and `verifyZero`.

Having a native support for constrains in the code enables translation to more efficient programs, since checking conditions using `if` statements is generally less efficient.

3.2.3 Type and Syntactic Constraint Checking

Introducing new types and features requires additional type rules and syntactic constraints to be enforced/checked by the front end. For example, we need to add type rules for checking bitwidth properties during assignments and operations. Syntactic constraints enforce the structure of the input program. We define the type system and syntactic constraint rules using a special language in the underlying framework. Note that our framework checks such rules and constraints interactively during development, which makes development easier.

3.2.4 Front-End Code Generation

In this section, we briefly discuss few technical points regarding the transformation from the xJsnark code to normal Java code. JetBrains MPS requires specifying Java code to replace the extension feature in some special language. Therefore, for every type or feature we have, there are Java classes in the back end that handle its functionalities.

To translate conditional `if` statements and other data-dependent constructs, one naive approach would be to compute the result of the conditional statement as

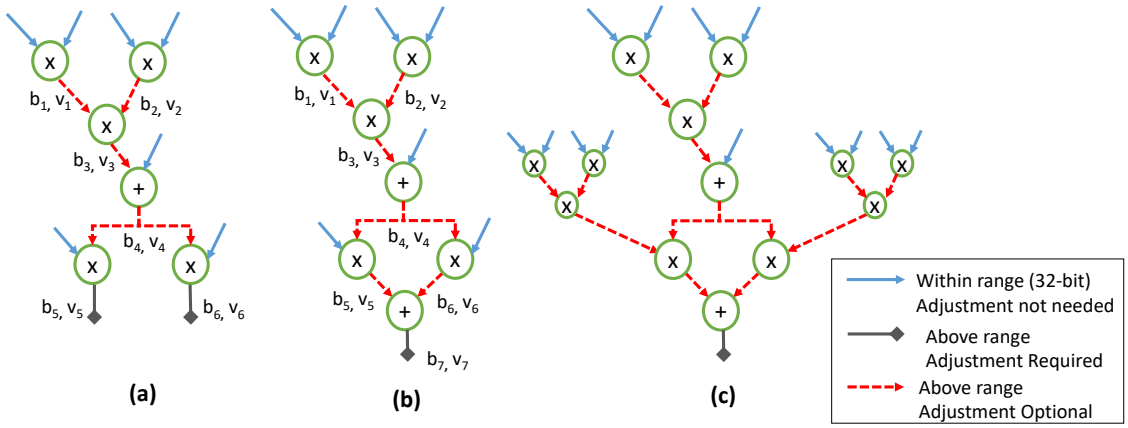

Figure 3.2: Bitwidth adjustment examples

a bit variable, multiply it by other bits from outer conditional statements (if any), and then use this bit variable in all operations, such as assignment operations, to ensure that all effects are applied only if this bit is true.

A more efficient approach would be to apply the single assignment algorithm, as in the Fairplay [22] compiler. This will help reduce the number of gates in nested if statements, and when multiple operations are inside a block. We adopt this approach in our implementation.

3.3 Data Type Representation

In this section, we describe how xJsnark's back end represents data types and implements their operations. The discussion will be mainly focused on integers and field elements (recall that integers are field elements, where the modulus is a power of two). In the beginning, we make a distinction between short integer and large integer arithmetic. Assuming the underlying SNARK prime is p, typically a 254-bit

prime, then short integer arithmetic is applied when the modulus of the field p' is less than $\sqrt{p}$. For simplicity, when $\lceil \log_2 p' \rceil < 0.5 \lceil \log_2 p \rceil$. The reason for that decision is to make sure that the initial result of multiplying two elements will be less than p, i.e. fits in one wire. Otherwise, dividing the element across multiple words will be needed, as will be shown when the long arithmetic is described in detail.

Representing the operations of a different field on top of the SNARK field requires some care, as operations are done modulo p in the circuits. Therefore, adding two 32-bit integers is expected to produce a 33-bit value, but we mainly care about the least significant 32-bit. Converting the 33-bit value to the correct 32-bit is an expensive process that requires 34 multiplications. This conversion is not always necessary. In other words, many operations can be done, e.g. additions or multiplications, before it comes necessary to convert the element to a value within its field, e.g. to avoid overflows (when the result of an operation exceeds p), or if the element is involved in a comparison. The same holds for general field elements, but the conversion is even more costly here due to a more expensive remainder operation, as will be shown shortly.

In order to make our implementation safe against overflows, we keep track of the maximum value that any element can have at any point. For an element or word x, we denote the maximum value it can have as x_{max}.

In the following subsections, we discuss different design decisions for both short and long integer arithmetic.

3.3.1 Short Integer Arithmetic

Assume the field being represented is $F_{p'}$, $\log_2 p' < 0.5 * \log_2 p$. Each element is represented as a single word.

3.3.1.1 Bitwidth Adjustment

Addition and multiplication are straightforward operations for short integer arithmetic, however they typically increase the bitwidth of the resulting element beyond $\lceil \log_2 p' \rceil$. Deciding when to convert an element back to the range of its field is a challenging problem. First, we have to make a distinction between three types of elements.

- Elements within range, i.e. $0 \leq e_{max} < p'$: examples include input elements (which are guaranteed to be in range), or elements resulting from bitwise operations, e.g. bitwise XOR. In that case, the output element is guaranteed to be within range, as it is been computed based on packing individual bits.

- Elements that could be above the range, i.e. $p' \leq e_{max} < p$ and **required** to be returned within range: This may include elements that are labeled as output, elements that are involved in bitwise operations and comparisons, elements involved in operations like division or remainder, and elements that are involved in memory operations. In the context of integer elements, this also includes elements that are involved in operations or assignments with higher bitwidth. For example, adding a 32-bit element to a 64-bit element, requires that the 32-bit element is in

range. Otherwise, this will lead to a wrong result.

- Elements that could be above the range, i.e. $p' \leq e_{max} < p$ but are not always required to be within range: This includes intermediate elements between multiplication and addition operations, such that none of the above conditions apply.

In order to be able to classify the elements into the above categories, we make an initial pass constructing a dummy circuit to identify the class of each of the elements. This is one main objective for the initial phase described in Figure 3.1. Based on the classification of the elements above, the main question is when to adjust the bitwidth of an element e falling in the third category, to achieve the following two goals:

- Ensure no overflows can happen in later operations involving e.
- Minimize the total cost resulting from adjustments overall the circuit.

This can be modeled as a constrained optimization problem. To illustrate that by example, Figure 3.2 provides sample circuits, assuming $p' = 2^{32}$. For each element e_i that does not fall in the first category, we define the following two variables b_i, v_i: b_i is a binary variable denoting whether e_i is going to be adjusted or not, while v_i represents the value of the bitwidth before applying adjustments if any. b_i is 1 for any elements falling in the second category. Note that adjusting the bitwidth of an n-bit element will cost $n + 1$ constraints. Now, we can specify both the objective function and the constraints, for circuit (a) in Figure 3.2.

The objective function can be defined as the total number of constraints resulting from all adjustments $f = \sum_i b_i(v_i + 1)$

Subject to the following constraints

$$v_1 = v_2 = 64$$

$$b_5 = b_6 = 1$$

$$v_3 = v_1 + b_1(32 - v_1) + v_2 + b_2(32 - v_2)$$

$$v_4 = v_3 + b_3(32 - v_3) + 1$$

$$v_5 = v_6 = v_4 + b_4(32 - v_4) + 32$$

$$v_i \in \{32, .., \lfloor \log_2 p \rfloor\}$$

$$b_i \in \{0, 1\}$$

It is possible to express the problem as a function of b_i's only, however, the reason v_i's were introduced is that the size of the expressions will grow without the equality constraints. Based on our experience trying multiple nonlinear optimization algorithms, using the above approach for large circuits will not be efficient, but has the advantage of producing optimal solutions.

Greedy Strategies. Due to the cost of the above solution, one alternative could be to apply a simple greedy algorithm after the initial phase, where adjustments are only introduced if the next operation is going to result in an overflow. This approach can work well for most of the applications we consider. Note that the initial phase itself can introduce some optimizations through the knowledge of how elements are being used later. For example, in this sample example, it can be noted that $x1$ is being used in a bitwise operation later, i.e. it falls under category

2 defined earlier, and its bitwidth will be adjusted in all cases. However note that the line $x2 = x1 * x1$ occurs before the bitwise operation. This line could make use of the fact that $x1$ will be adjusted back to its range. This is not possible unless we make a complete pass over the program first as we do already in the first phase in the back end.

```
// assume in1, in2 are uint_32 variable inputs, while out is uint_32 output.
uint_32 x1 = in1*in2;
uint_32 x2 = x1*x1;
uint_32 x3 = x1 ^ in1;
..
out = x2+x3;
```

Another greedy approach will be to study how an element contributes to different paths leading to an adjustment in the end. Note that solving the above optimization problem (Figure 3.2 [a]) will lead to $b_4 = 1$, while $b_1 = b_2 = b_3 = 0$. This result can possibly be justified by noticing that wire #4 contributes eventually to two distinct paths through two multiplication gates, each leading to an adjustment in the end. However, in [b] it is expected that no adjustment will be needed in any of the intermediate levels, although wire #4 still contributes to two paths, but they are not leading to different adjustment outcomes. It's possible during compilation time to study which intermediate wires contribute to the end points, and select wires that contribute with multiplications in more than one path. However, this won't ensure optimality in all cases as well. It is also still important to handle the possibilities of overflows, as the strategy does not directly take that into account.

For example, in Figure 3.2 [c], applying the strategy above directly might lead to an overflow in the lower level, except if the corresponding wire is adjusted.

In summary, the greedy strategies will not always achieve the optimal solution, but can compile applications faster.

Adjustment Implementation Adjustment for an element x is done by computing the element $r = x \bmod p'$. Implementing the remainder operation is straightforward when p' is a power of two. In that case, it is enough to split x, trim the unnecessary bits, and pack the rest to a new element (if needed). The cost for the adjustment in that case is nearly: $\log_2 x_{max} + 1$ constraints.

To get the remainder in the case of general field elements where p' is not a power of two, we use the power of SNARK verification where the prover can provide two values r and q, and the circuit checks the following constraint: $qp' + r = x$, while restricting the bitwidth of q such that no overflow will happen, and also asserting that $0 \leq r < p'$. This last constraint is implemented by checking that the bitwidth of r is less than or equal to the bitwidth of p', and applying the comparison test mentioned earlier to ensure that $r < p'$.

However, an additional optimization in the case of fields, where p' is not a power of two, is to implement bitwidth adjustment differently for the elements falling in the third category described earlier. Such elements do not have to apply the second part of the third constraint. It is enough for the prover to provide a value r that satisfies the bitwidth constraint, but no need to check that $r < p'$, as adjusting the elements of the third category is mainly done to avoid overflows.

3.3.1.2 Subtraction

To subtract two short elements x and y, the result will depend on p if $x < y$. To avoid that, we introduce an auxiliary constant a such that $a = c.p'$, where c is the smallest integer such that $c.p' \geq y_{max}$. If $c.p' \geq p$, this means that the value of y needs to be adjusted, i.e. we need to compute the value $y \bmod p'$ in the circuit as above, and set c to 1 for the subtraction. The result of the subtraction will be: $a + x - y$.

3.3.1.3 Division and Remainder operations

Division and remainder operations supported only for integers have a similar implementation to the implementation of bitwidth adjustment described earlier. A similar approach can also apply for the multiplicative inverse for field elements, by forcing the remainder of the product of the operand and the result to be equal to 1 (mod p'), while checking that the inverse result is in range.

3.3.2 Long Integer Arithmetic

Typically, the prime field used in zk-SNARK implementation is a 254-bit prime field, however in many cryptographic applications, longer integer arithmetic is required for high security, such as in RSA or Elliptic Curves. Hence, a long integer is represented using a group of wires rather than a single wire. An n-bit long integer x is represented by a group of b-bit words $x[i]$, where $i \in \{0, 1, ..m-1\}$, and $m = \lceil \frac{n}{b} \rceil$, such that $x = \sum_{i=0}^{m-1} x[i]2^{ib}$, and $b < \log_2 p$.

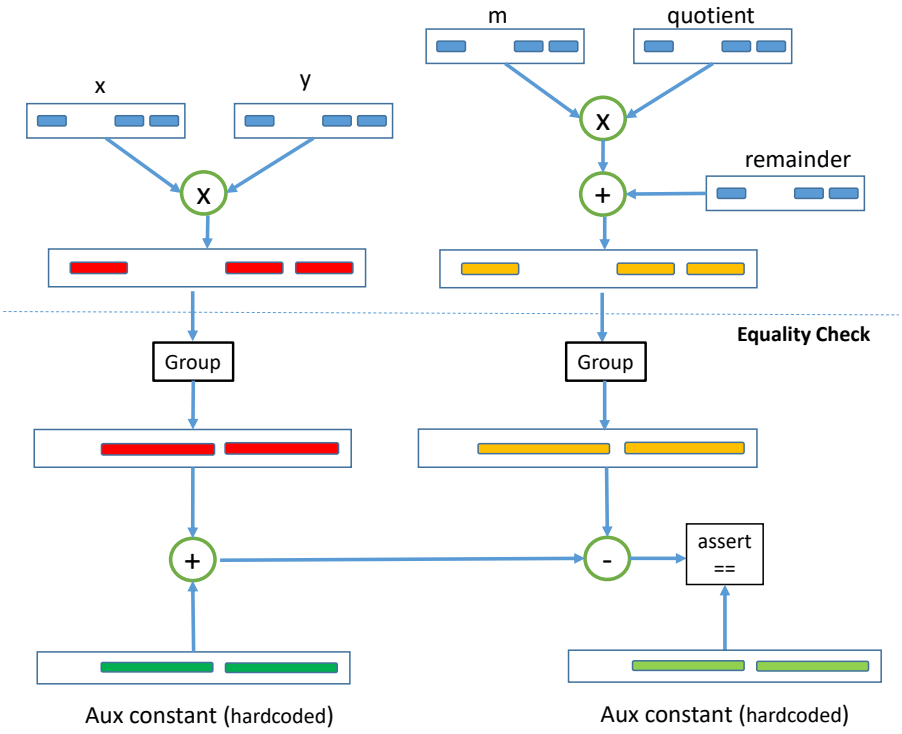

Figure 3.3: Equality assertion in a modular multiplication circuit. Auxiliary constant values are carefully chosen in the last step to facilitate the verification.

A technical question here is how to set b properly to achieve high performance. As we are going to show, setting the bitwidth b, to be the largest possible while avoiding overflows does not necessarily result in the best performance or the cheapest circuit.

Additionally, as in the short integer case, care is also needed when another field is represented on top of the 254-bit field case. However, unlike the short integer case, the algorithms for when to adjust long integers back will have to be adapted a little bit as in the following. Most of the operations are similar to what have been

discussed in the short integer case. We mainly highlight the major differences.

3.3.2.1 Multiplication

Given two long integers x and y each is m words. While addition is straightforward, i.e. can be implemented by adding corresponding chunks, multiple options exist for multiplying x and y in the circuit. For example, we can either apply the trivial $O(m^2)$ approach, where $z[i] = \sum_{j+k=i} x[j]y[k]$, or Karatsuba's method [45], which costs $O(m^{1.58})$ multiplications.

However, it's possible to have an $O(m)$ approach. The result of the multiplication z can be computed independently by the prover and provided as a witness to the circuit. Then the circuit can verify the result using the following approach: For each $c \in \{0, .., 2m-2\}$, the circuit checks this constraint: $(\sum_{i=0}^{m-1} x[i]c^i).(\sum_{i=0}^{m-1} y[i]c^i) = \sum_{i=0}^{2m-2} z[i]c^i$. In other words, the prover will be required to provide $2m-1$ values that satisfy a linear system of $2m-1$ equations, that has a single solution. The total number of constraints to implement this verification circuit is $2m-1$. Note that we mainly rely on the observation that multiplication by hard-coded constants in the circuit is almost free. Figure 3.4 illustrates a comparison between the three approaches above with respect to the proof time on a single processor.

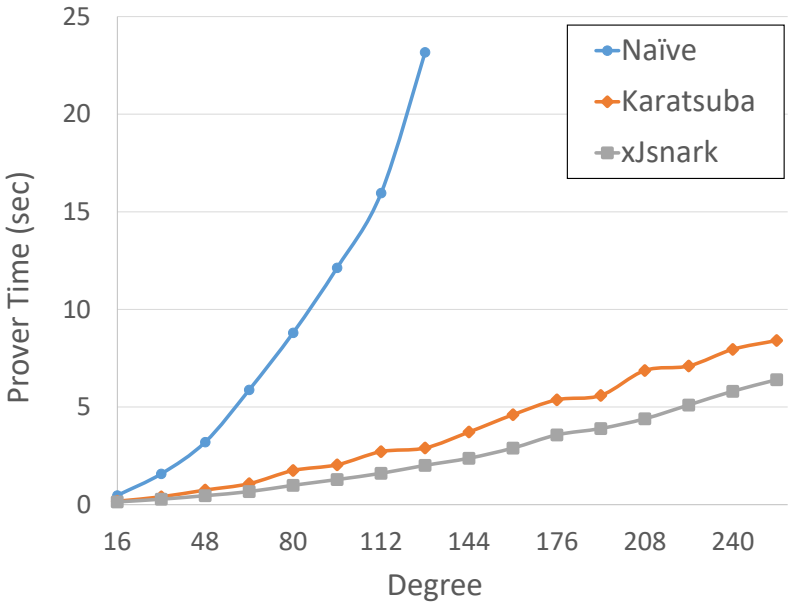

Figure 3.4: Long integer multiplication methods

3.3.2.2 Subtraction

Subtraction in the case of long integer arithmetic is more challenging. Recall that in the case of short integer arithmetic, an auxiliary constant value was added in order to make sure the result stays in range. In the case of long integer representation, we also need to ensure that the result of subtracting corresponding chunks stays in range. The way we do this is as follows: To subtract two long elements x and y, we introduce an auxiliary constant a such that $a = c.p'$, where c is the smallest integer such that $c.p' \geq y_{max}$. Additionally, it must be possible to represent $a = c.p'$ as a group of words $a[i]$, such that $a[i] \geq y[i]_{max}$ for all i.

3.3.2.3 Bitwidth Adjustment

Similar to the case we had before, bitwidth adjustment in the case of long integers is needed if any of the operations involving its words may overflow, or if the number is being used for comparison or equality checks.

The same greedy procedures described earlier can be applied in the case of long integers, however an easy observation to make is that when a long integer is involved in a multiplication, this means that each of its words contributes to multiple words in the output number, which implies the involvement in multiple bitwidth adjustment end points. This can make the decision of adjustment more straightforward. We apply this simple heuristic: we adjust any long integer that is an output of a long integer multiplication before being involved in another **multiplication** operation.

3.3.2.4 Equality Assertion

In many applications such as RSA or Elliptic Curves, many inverse and remainder operations will be required to verify the correctness of the results. These operations require applying equality constraints on long integers. This is the most expensive part in the circuit, as in [12, 16].

What makes the problem of equality assertion in long integers more challenging is that the words of a long integer operand may not be bounded to their starting range. For example, consider the main building block of modular exponentiation illustrated in Figure 3.3. Both of the integers $z_1 = xy$, and $z_2 = nq + r$ are supposed to be equal, but their words do not have to be equal, as any $z_1[i]$ or $z_2[i]$ are not

expected to be within the range $[0, 2^b - 1]$. Assuming x and y had properly bounded words, then it's expected that $z_1[0]$ falls in the range $[0, 2^{2b} - 1]$ for example. The way to force equality efficiently would be by noting that the first b bits for $z_1[0] - z_2[0]$ must be zero, while the rest can be propagated to the check done at the second word, in which the first b bits of the two words will be checked as well, and so on. This is the approach applied adopted by [12, 16]. This costs about b constraints per each pair of words, resulting in a total of $2mb$ gates approximately (as xy requires $2m - 1$ words).

Additional optimization over [12, 16] is to utilize that addition and multiplication by constants are free operations. So, instead of forcing the first b bits of $z_1[0] - z_2[0]$ to be zero, we can instead apply a grouping stage (as far as b allows). For example, if $b = 64$, it's clear that we can apply this constraint instead: force the first $2b$ bits of $z_1[0] + 2^{64} z_1[1] - (z_2[0] + 2^{64} z_2[1])$ to be zero, propagate the rest of the bits to the next check (nearly b bits). This implies that the circuit will need to pay about 64 gates per each 2 pairs of words. If b was chosen the highest possible (e.g. 120 in Cinderella's implementation [9]), the number of words will be less by a factor of 2, but the cost will be higher per every pair of words. As b decreases, the more grouping that can be done, and the more savings. That said, decreasing b results in a higher number of words, which means more cost for the multiplication module in the earlier steps. Based on a parameter exploration for RSA 2048, choosing b at about 32 bits provides much more savings compared to both extremes (Figure 3.5).

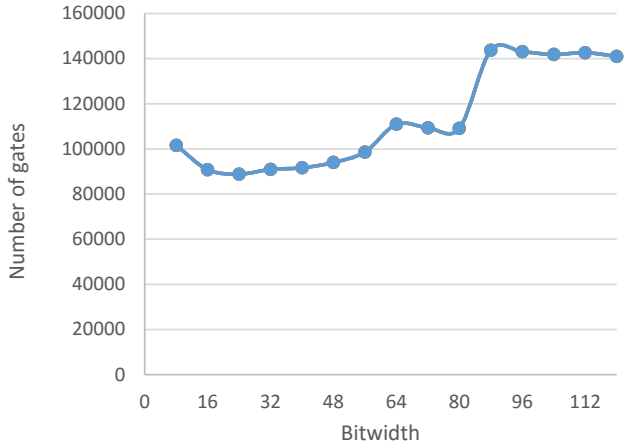

Figure 3.5: Bitwidth Effect on Number of Gates in RSA modular exponentiation circuit

3.4 RAM Implementation

One challenge in translating programs to circuits is that programs make *dynamic* memory accesses (whose addresses are not known at compilation time), whereas circuits have *static* wiring. Before presenting our approach, we overview existing techniques for verifying dynamic memory accesses [4, 11]:

Linear Scan. Each memory access is performed through a linear sweep of the entire memory. Roughly speaking, this costs $O(kn)$ for making k memory accesses where n is the total memory size.

Merkle Tree. Memory accesses are verified through memory checking techniques such as a Merkle hash tree. This approach requires roughly $\Theta(k \log n)$ hash computations inside the SNARK circuit for making a total of k accesses. Although the

dependence on n is logarithmic, the hash evaluation is expensive, and therefore this approach is in practice inefficient unless for very large choices of n.

Permutation Network. Memory accesses are verified using an AS-Waksman network [57]: This approach costs $O((k+n)\log(k+n))$ for k accesses where the starting memory size is n. This approach was proposed in [24], and subsequently used in TinyRAM [25], and Buffet [11].

In the remainder of this section,

- We start by revisiting the earlier random memory access methods, and discuss some low level optimizations.
- We propose an additional random memory access algorithm that can do potential savings in many scenarios where the memory being accessed is hardcoded and read-only, such as in cryptographic S-boxes.
- Finally, we put all together and discuss how the smart memory is implemented.

3.4.1 Improvements for earlier methods

3.4.1.1 Merkle tree approach

The main bottleneck in Merkle tree implementations is the cost of the hash function applied at each level. Pantry reported about 4700 multiplication gates per level. Instead, it is possible to use a SNARK-friendly collision resistant hash function, as the one initially proposed in [21], and later analyzed in [16]. Using such hash function, the cost per level can be 2032 gates to achieve more than 128 bit

security level.

3.4.1.2 Permutation network approach

We describe a low level optimization for the permutation network used in previous work to implement general random memory accesses.

A permutation network is typically implemented as an AS-Waksman network [57] in order to fit the arbitrary number of accesses. Buffet reported the cost per access nearly to be: $c + 10 \log k + 2 \log n$, where k is the number of memory accesses, n is the memory size and c is a constant. The reason for the factor of 10 is due to the observation that every memory access contributes a record of four wires to the permutation network, since every memory access is implemented as a tuple of four elements (Timestamp, Index, Data Element, LOAD/WRITE). Any switch in the permutation network will receive two tuples as input, and a verifiably binary input to set the direction of the switch. The first improvement will be to reduce the factor of 10 by half, by observing the linear dependencies between the switch outputs.

Furthermore, in some situations where the number of memory accesses is high compared to the memory size, such as in small memories and short data elements, it will be better to pack the four elements of a tuple together to a single wire, such that every switch in the network will only have two wires as inputs. An interesting observation here is that a switch in that case can be implemented without using an input to handle the switching. In fact, it can be implemented using one constraint.

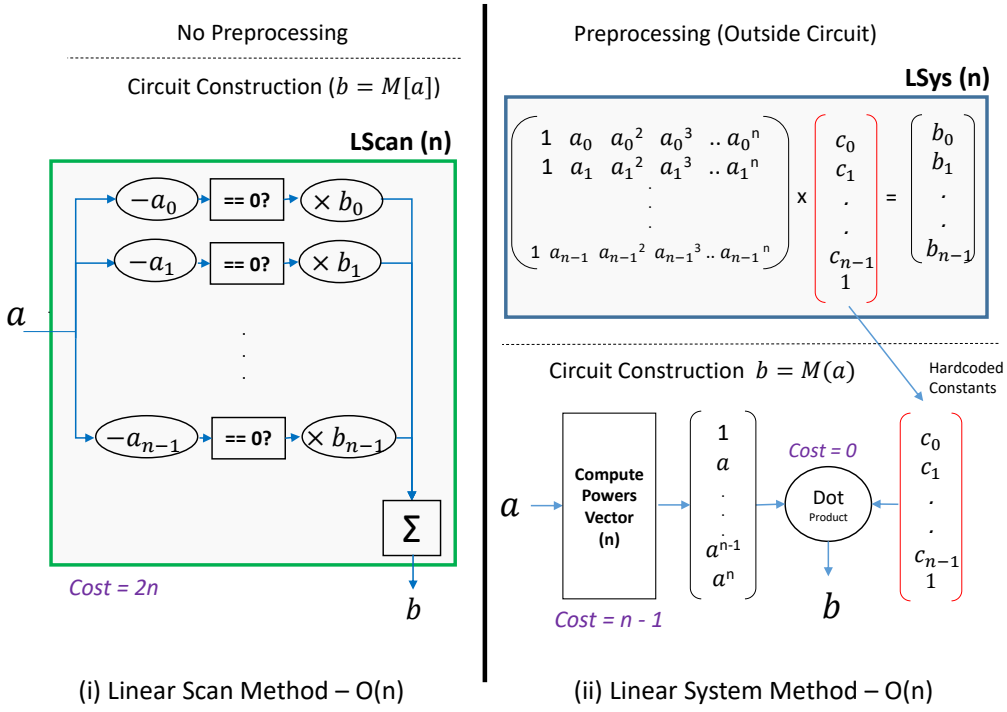

Figure 3.6: $O(n)$ methods for read-only memory access

For a switch receiving two wires w_1 and w_2, the prover provides the first output wire as an external witness w'_1, such that $(w_1 - w'_1)(w_2 - w'_2) = 0$, and the other wire can be computed as a linear function of the three other wires, simply by $w'_2 = (w_1 + w_2) - w'_1$. Deciding whether to do the packing or not is a decision by the compiler that depends on the memory workload, and the type/size of the data elements stored.

Further improvements to the permutation network approach in special cases is ongoing work.

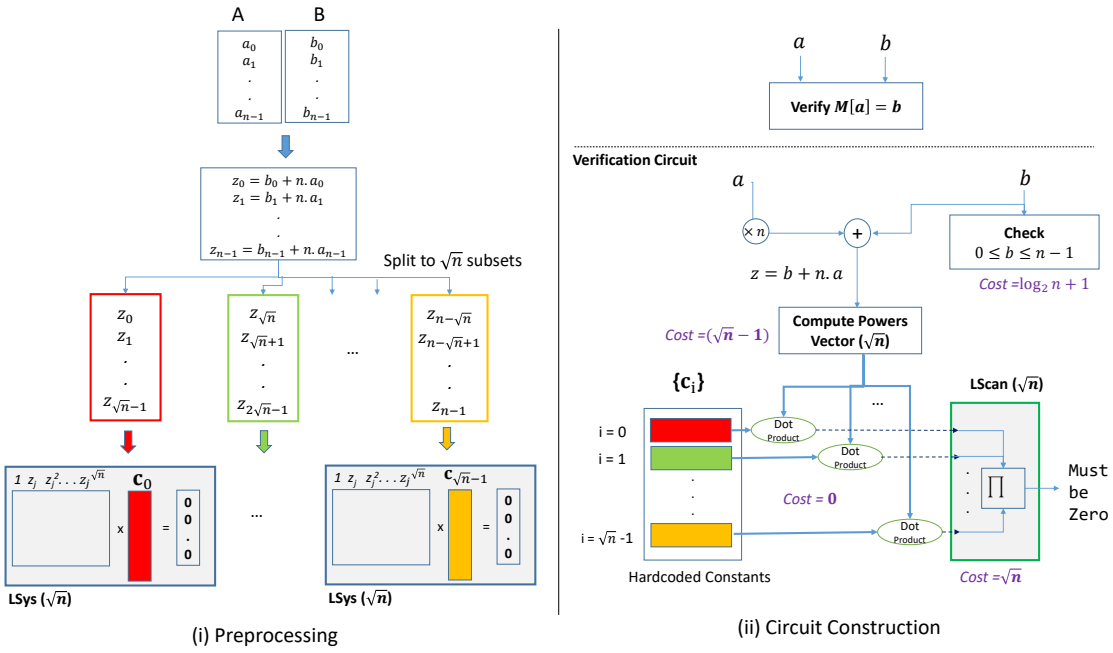

Figure 3.7: New approach for static read-only memories

3.4.2 Algorithm for static read-only memory access

In this section, we discuss a new method for implementing dynamic memory accesses in read-only arrays whose contents are prepopulated and never going to change. This can fit many applications where the memory content is known in advance, such as the cases for S-box evaluation in cryptographic primitives, as in AES. It can also be extended to other cases where look-up tables are used instead of expensive floating-point arithmetic computation, as in logarithmic functions.

Problem statement and known techniques. Formally, given a mapping M from $A = \{0,1\}^{\log_2 n} \rightarrow B = \{0,1\}^{\log_2 n}$. Assume there is no straightforward mapping from A to B, as in the case of random permutations. For simplicity, assume that n is an even power of two. The mapping is fixed and known in advance, how-

ever, an accessed index a in the circuit is unknown at compilation time. To resolve the mapping and obtain $b = M(a)$ in the circuit, we can employ any of the known techniques, including linear scan, Merkle tree, or permutation networks.

Our algorithm. Now, we will show how to have a useful $O(\sqrt{n})$ algorithm for resolving $b = M(a)$ that can be better than all the above cases in practical cases.

Building block: polynomial function via a linear system solution. We first describe a building block for accessing a read-only memory. Although this building block alone requires $O(n)$ cost per access, we will later explain how to combine this technique with our naïve linear scan algorithm, to obtain a new $O(\sqrt{n})$ algorithm.

An n-degree polynomial function can be introduced to obtain a relation between the inputs and the outputs. In a preprocessing phase, the compiler constructs an $n \times (n+1)$ matrix P, where each row is the power vector $[1, a, a^2, .., a^n]$ for all $a_i \in A$, and a column vector $\mathbf{b}$ that has the corresponding n elements of B. Then, a coefficient vector $\mathbf{c}$ with $n+1$ elements can be obtained by solving the linear system $P\mathbf{c} = \mathbf{b}$. Note that the last element of $\mathbf{c}$ is set to 1. A solution will always exist since the finite field we operate on has a prime order. Then, in the circuit, the coefficients $\mathbf{c}$ can be just hardcoded as constants in the circuits, and to resolve an index a, the power vector $\mathbf{a}$ is constructed costing $n-1$ gates, and the result is obtained by the dot product $b = \mathbf{a} \cdot \mathbf{c}$ costing zero gates. We denote this method as the linear system-based method in the next discussions. (Although there can be more efficient methods to compute the coefficients, we keep the linear system naming as we rely on linear systems eventually in our last optimization).

Intuition. The proposed technique relies on non-determinism. The actual value of $b = M(a)$ does not have to be computed by the circuit, but instead, the prover can provide b as a witness, and the circuit can just verify that the pair (a, b) is a valid pair with respect to M.

The $O(\sqrt{n})$ method we propose for checking the validity of the pair is a hybrid of the two $O(n)$ methods mentioned earlier (Both methods are illustrated in Figure 3.6). The approach is mainly inspired by two observations in the second method: 1) The cost of the dot product operation is zero, since the coefficient vector is computed in advance. 2) The cost of the technique is mainly due to computing the powers of a, which costs $O(n)$ multiplications.

Now, the goal is to reduce the length of the power vector, while introducing multiple free dot product operations instead. In brief, this will be done by decomposing the problem of accessing one array that has n distinct elements to checking membership in $\sqrt{n}$ arrays, each has $\sqrt{n}$ distinct elements. In particular, for each array, a linear system is solved in the preprocessing phase, and a coefficient vector is obtained. Then, in the constructed circuit, a shorter power vector is computed (only up to $\sqrt{n}$ elements), and then the free dot product operations are applied on the $\sqrt{n}$ hardcoded vectors. The output element will be verified using a more efficient version of the linear scan method, which will iterate only over $\sqrt{n}$ elements instead of n elements.

Approach. More formally, during the compilation time, since the memory is fixed, the back end can compute the set $S = \{z_0, z_1, ..z_{n-1}\}$, where $z_i = b_i + n.a_i$. Note that the elements z_i are guaranteed to be distinct even if the values b_i are not, as

the indices a_i are distinct, and $0 \leq b_i < n$.

The back end then divides the set S into $\sqrt{n}$ subsets, such that each subset $S_j = \{z_k : j\sqrt{n} \leq k \leq (j+1)\sqrt{n} - 1\}$ for all $j \in 0, 1, .., \sqrt{n} - 1$. This implies that the cardinality of each S_j is $\sqrt{n}$. For each S_j, the back end constructs the following linear system of equations: $\sum_{k=0}^{\sqrt{n}-1} \mathbf{c}_{jk} z^k + z^{\sqrt{n}} = 0$ for each $z \in S_j$, where $\mathbf{c}_j$ is a column vector associated with S_j. Since every set contains $\sqrt{n}$ distinct elements, it's expected to have $\sqrt{n}$ equations per each linear system, and a unique solution always exists since we operate in finite field with a prime order.

In the circuit construction phase, the back end hardcodes the vectors $\{\mathbf{c}_i\}$ in the circuit. To resolve a random access to index a, the prover provides a witness b, and the circuit checks that $b = M(a)$. In other words, the circuit checks that the value $z = b + a.n$ belongs to S. First, the circuit checks the range of b, i.e. $0 \leq b < n$. This costs about $\log_2 n + 1$ gates. Then, the power vector $\mathbf{z} = [1, z, ..., z^{\sqrt{n}}]$ is computed (costing $\sqrt{n} - 1$ gates), and applied to each vector in the set $\{\mathbf{c}_j\}$ via free dot product operations. If the value b provided by the prover is correct, then the value z should belong to only one of the sets S_j, and result in a zero value in the corresponding dot product operation. To verify the correctness of b, it suffices to check that **any** of the dot product outputs is zero. This can be done through a more efficient linear scan path that just multiplies all the values and asserts the product value to be zero. This costs only $\sqrt{n}$ gates. An additional visual illustration of the method can be found in Figure 3.7. The actual cost in the circuit will be equal to $2\sqrt{n} + \log_2 n$ constraints per access. Further optimization is described shortly.

Security. Note that the preprocessing of the above scheme is equivalent to finding

polynomials at specific roots (Although that would be more efficient, we keep the linear system notation for further optimizations that are discussed shortly). To see how the scheme above protects against the case where a prover claims that b' is the result of $M(a)$, while $b' \neq M(a)$. Recall that only the permissible set of values (roots) are of the form $z_i = b_i + n.a_i$. For a false mapping, the following condition will need to hold $b' + n.a = z_j = b_j + n.a_j$ for some j, i.e. $b' = b_j + n.(a_j - a)$. There can be two cases here: if $a_j = a$, then $b_j = b'$ (contradiction). If $a_j \neq a$, then b' will either satisfy $b' \geq n$ or $b' < 0$, which contradicts the range check applied by the circuit: $0 \leq b < n$, assuming n is much smaller than the prime order p as in practical scenarios.

Further Optimization. Additional optimizations can be made to the earlier approach to reduce the number of gates per access by half, but still within the $O(\sqrt{n})$ complexity. For example, in the above description, instead of completely relying on the power vector in constructing the linear systems, if the bit decompositions of a and b are available/needed for other purposes in the circuit (b's bit decomposition is already needed for the range check), then the bits can be used instead to partially construct the linear systems. This has to be done *carefully*, such that the prover cannot cheat. In our implementation, this may require shuffling the elements before dividing them into $\sqrt{n}$ groups, and including few elements from the power vector while checking the rest of the restricted domain to verify that the prover cannot cheat.

Such optimization helps in reducing the cost of small memory access, as in the AES implementation, since the bit decomposition of a and b are typically needed

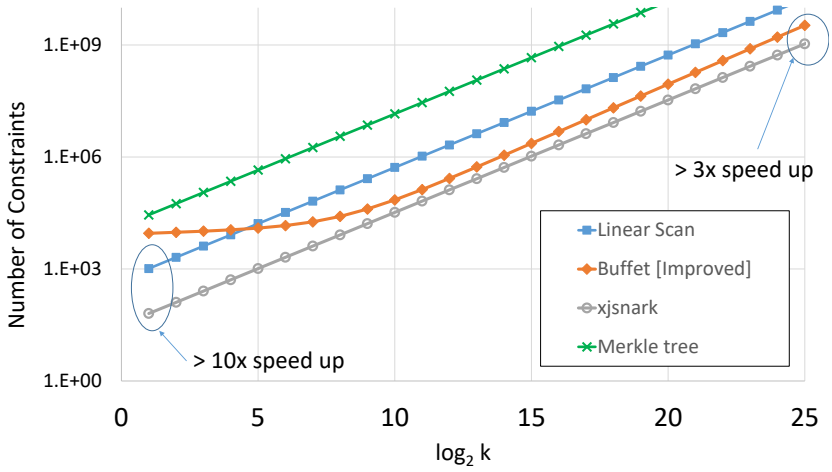

Figure 3.8: Comparison between the proposed $O(\sqrt{n})$ method for read-only memory access, and other optimized approaches when $n = 256$. k represents the number of accesses.

for other parts in the circuit.

Comparison with earlier methods In case of small hardcoded memories, such as in AES S-box (which is a 256-element array), the proposed method is clearly better than the $O(n)$ approaches. Additionally, it's much better than the Merkle tree approach due to the large cost of the hash function. When the number of accesses is high, the main competitive to our approach in the case of small memories is the permutation network approach, which has two main issues: 1) Since the memory does not start empty, n write operations will need to be inserted in the permutation network as input initially. 2) Additionally, the total cost of applying the permutation network is $O((n+k)(\log(n+k)))$ as mentioned earlier, which means it also depends on the number of accesses made, besides the memory size. Table 3.1 compares all the techniques discussed so far.

Table 3.1: Comparing read-only constant memory access techniques in terms of the total number of constraints for all accesses (n denotes the memory size, and k denotes the total number of reads.)

	Total Cost (Complexity)	Actual Total Cost
Linear Scan	$O(kn)$	$2kn$
Linear System	$O(kn)$	kn
Merkle Tree	$O(k \log n)$	$2000k \log_2 n$
Buffet Perm. N/w	$O((n+k)(\log(n+k)))$	$(n+k)(\log_2(n+k))+$ $2\log_2(k+n)+3\log_2 n$
xJsnark	$O(k\sqrt{n})$	$k(2\sqrt{n}+\log_2 n)$

Case study when $n = 256$. Figure 3.8 compares the existing approaches (after optimizations) to the proposed $O(\sqrt{n})$ method (in a logarithmic scale), identifying in which regions each algorithm performs better. As shown, the proposed algorithm performs better than all the other alternatives, achieving speed-ups ranging from more than 10× when the number of accesses is 2, to more than 3×, when the number of memory accesses is more than 32 million.

3.4.3 Smart Memory

In our framework, a programmer will use a special syntax to instantiate a smart memory, however the programmer will be able to use the typical array operators. In the first preprocessing stage of the back end, each memory is studied separately, and the compiler takes the following factors into account:

- The number of read/write operations.
- The type/size of data being accessed.
- Whether the memory contents is read only and known in advance or not.

Based on these factors, the back end decides the most appropriate implementation, and its specifics. For example, in case of a general read-write memory (with contents unknown during compilation time), it can decide that a linear scan method is better than constructing a permutation network, when the operations done are not many or when they involve few random accesses among many accesses to constant locations. Also, in the case of read-only hardcoded memories, the framework automatically chooses the best implementation, and performs any required preprocessing.

3.5 Arithmetic Optimization Module

In the previous optimizations, we discussed how to reduce the number of constraints resulting from split gates, random memory accesses and other operations. In this section, we describe a low-level optimization that can further reduce the number of gates via *multivariate polynomial minimization*.

This module is motivated by the following: As mentioned earlier, the cost for bit-level operations is high. Any inefficient implementation of Boolean operations will have an effect on the size of the circuit that correlates with the bitwidth of the variables. For example, in this SHA-256 code, the majority variable is being computed as in the following equation, where all a, b and c are 32-bit words.

```
for(int i = 0; i < 64; i++){
  // ..
  maj = (a ^ b) & (b ^ c) & (a ^ c)
  // ..
  c = b;
  b = a;
  a = /* Code omitted f(maj) */;
}
```

If this equation is translated into a circuit directly, given the bits of a, b and c, computing each bit in maj will cost 5 multiplications per bit, however using minimization techniques, this can be reduced to 2 multiplications, saving a total of 6144 multiplications across all bits in all rounds. To achieve that, each bit i of maj can be expressed as:

$$t_i = a_i b_i; \quad maj_i = t_i + c_i(a_i + b_i - 2t_i)$$

This cannot be specified directly in high-level C or java, but instead, taking Geppetto as an example, the compiler supports special instructions to have access to bits, and to write constraints accordingly. An additional optimization that can be done is to observe that the variable b is assigned to c, and a is assigned to b. This implies that the maj computation across rounds will have shared variables on the bit level. Making use of that observation, additional 1024 multiplications can be saved.

To perform such optimization automatically, we implemented a customized technique for multi-variate polynomial minimization based on [58] as a building block. This block takes a set of multivariate polynomials as inputs, and tries to minimize the expressions cost based on a greedy strategy. Due to the large circuit

sizes, we developed techniques for clustering the arithmetic expressions into smaller subgroups that can be optimized independently in parallel.

In the following subsections, we describe the steps in detail.

3.5.1 Assignment of Input and Output Symbols

In many programs, it is not possible to express the circuit outputs as polynomial functions of the inputs. This is mainly due to having special kinds of gates where outputs cannot be written as polynomial functions of the inputs. This includes the split gate, the zero checking gate, and typically user-defined gadgets that rely on verification properties, e.g. a gadget for verifying a linear system of equations. Such gates appear in any programs that have bitwise operations, conditionals, division and others. Therefore, we may need to split the circuit to multiple sub-circuits depending on its shape. The way this is done is by labeling wires as **opt-input** (denoting an input variable to an optimization problem) or **opt-output** (denoting an output variable to an optimization problem) in an initial phase. The notion of opt-input and opt-output variables used above should not be confused with the input and output wires of the circuit. After labeling, the sub-problems are chosen accordingly.

The criteria by which we initially label wires as opt-input or opt-output variables, are as follows.

- Program input and prover witness wires are labeled as opt-inputs, while outputs are labeled as opt-outputs.

- For any gates in which the output cannot be expressed as a polynomial of the input, the inputs to the gate are labeled as opt-outputs, while the outputs of the gate are considered opt-inputs to be used in later expressions. This applies to the split gate and conditional gates. Furthermore, although the pack gate does not fall under the same category (as its output can be expressed as a linear combination of its inputs), we apply the same rule here in order to separate the Boolean operations from arithmetic ones.

- All inputs to assertions, which have no output wires, are labeled as opt-outputs.

Additional criteria can also be employed for selecting opt-output wires. If an expression gets large, we can split the circuit at that point, and introduce a new opt-input. Another approach could be to rely on the usage count. When the usage count of a certain intermediate wire is high, this may suggest that this is a good point to split this part of the circuit. For example, assume a program that computes a linear function of the inputs, and then use the result in a heavy computations later that are independent from the previous part. To reduce the running time of the optimizer, it may be beneficial to use such criteria.

For the rest of the discussion, we will denote opt-input and opt-output wires assigned to the optimization problems as x_i and y_i. Figure 3.9 illustrates an example of how the wires of a simple circuit are labeled.

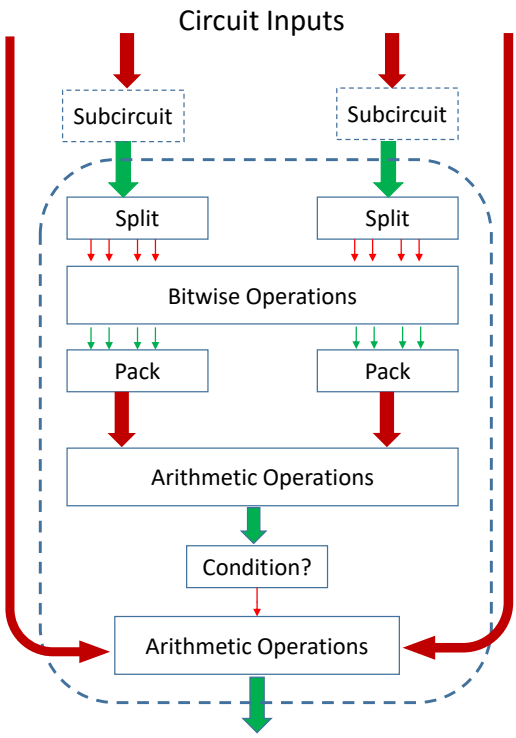

Figure 3.9: An example for how opt-input and opt-output wires are selected for multivariate minimization. Red wires indicate opt-inputs, and green wires indicate opt-outputs.

3.5.2 Clustering Expressions

After opt-input and opt-output variables are chosen, the expressions are computed by iterating over the gates of the circuit, and computing the output polynomial of each gate given its input polynomials.

Optimizing single multivariate expressions alone may not lead to the optimal solution. However, when we have a group of such expressions, we can eliminate shared computation and allow one expression to benefit from intermediate variables of another. For example, in the following case, the terms x_1x_2 and x_1x_3 can be

computed once, and no additional multiplications will be needed to compute any of y_i.

$$y_1 = x_1x_2 + x_1x_3; \quad y_2 = x_1x_2 + x_4; \quad y_3 = x_1x_3 + x_5$$

If each expression is optimized alone, the resulting expressions will be:

$$y_1 = x_1(x_2 + x_3); \quad y_2 = x_1x_2 + x_4; \quad y_3 = x_1x_3 + x_5$$

This will be more costly than the earlier case when we had a more global view of other expressions. Therefore, in order to decide whether an optimization is useful to apply or not, its effect on other parts of the circuit should be considered, by studying multiple related expressions at the same time.

Our approach is to instead cluster the expressions together based on the variables they share. We define a cluster as a set of expressions in which any two expressions must share at least two input variables in a term, or different power terms for the same input variable.

It should be noted that before running the next step, the symbolic evaluation of the circuit so far can help reduce the number of multiplication gates. For example, it can detect the cases where some operations are unnecessary, e.g. when a programmer writes code for a swapping operation using XOR instructions instead of using a temporary variable. Using the XOR method is much more expensive for SNARKs, compared to the free assignment instructions. The symbolic execution can detect and partially optimize this case.

3.5.3 Minimization

After clustering the equations based on the input variables, we implemented a customized optimization technique for reducing the cost of multivariate polynomial evaluations. Our implementation follows the greedy algorithm specified in [58], which is already based on known techniques in Multi-level logic synthesis, such as [59]. The implemented techniques can provide better results in comparison with multi-variate Horner's rule, and techniques for common sub-expression elimination. The main difference between our implementation, and the algorithm in [58] is that we distinguish between multiplication of variables, and multiplication by constants, to suit the cost model described earlier in Section 2.1.1.

3.5.4 Limitation

Our approach is greedy and does not guarantee optimality — in general achieving optimality is intractable. However, it was observed that this approach performs better than others for common cases [58]. Another limitation is its running time and memory consumption for large problems, therefore, we restrict the size of the problems tackled by this module, and we are working on optimizations.

3.6 Experimental Evaluation

In this section, we illustrate how our framework provides savings for multiple cryptographic building blocks, spanning hash functions, signatures, and encryption, compared to other compilers, while achieving programmability. Additionally, we dis-

cuss savings for random memory access. Furthermore, the evaluation also includes the full large circuit used by ZeroCash [7] for anonymous transactions, which we compare to existing manual optimized implementations, and show that our framework provides competitive performance to manual implementation, while reducing the programmer's effort.

3.6.1 Cryptographic Primitives

In the following, we evaluate four cryptographic primitives using our proposed framework and algorithms. The comparison is primarily done with respect to the state-of-the-art compilers, [11, 12]. The savings are measured in the number of the constraints (multiplication gates), while any additional programmer effort/experience required by the other compilers is mainly characterized by the following: 1) Introducing additional prover inputs and constraints to the circuit. 2) Specifying where bitwidth adjustment/remainder operations are needed. 3) Adding special procedures, e.g. a linear search code to implement random access.

SHA-256. We start by evaluating the SHA-256 circuit generated by the three compilers. SHA-256 has been used and optimized for zk-SNARKs in many earlier systems before, e.g. ZeroCash [7] and Hawk [2], mostly in a *manually* optimized way built using either libsnark [15] or jsnark [14], which provide a circuit that has approximately 27100 and 26100 gates respectively. In the following, we show how xJsnark reduces the gap between the automated solutions and the manual ones.

The code tested for SHA-256 is a typical SHA-256 code, except that Java

integer type is replaced by xJsnark's parametrized type `uint_32`. We assume a corresponding C code for both Buffet and Geppetto. We assume that the circuit hashes one block only, and that all inputs are variables, i.e. no padding is applied. Our experiments (Table 3.2) indicate that the SHA-256 circuit produced automatically by xJsnark achieves (**1.5**× and **1.7**×) savings over the alternatives. Two main reasons behind the savings in our automatically produced SHA-256 circuit. The first is the smart bitwidth adjustment, which saves about 3,200 constraints over Geppetto, 10,000 constraints over Buffet, and the multivariate polynomial minimization, which saves 8,800 constraints over both compilers.

Note that it is possible to enhance the SHA-256 circuits in Geppetto and Buffet, but with the cost of additional programming effort/experience (e.g. optimizing the expressions by hand in Geppetto, or adding casting statements in Buffet). In this example specifically, we assumed almost the same code in all of the three alternatives.

SWIFFT hash function. The SWIFFT function is a lattice-based hash function [60], in which the computations run in a field with $p' = 257$. As mentioned earlier, xJsnark allows the programmer to define Field types for arbitrary p'. On the other hand, Buffet and Geppetto do not have native data types that represent fields. As indicated in Table 3.2, xJsnark achieves the most savings while being easy to program. The savings are due to efficient remainder checking when the represented p' can be expressed as $2^n + 1$, while the programmability is mainly due to that fact that the programmer in that case does not choose where to do the remainder

operation. In comparison, Buffet supports mod operations (in a less efficient way), and the programmer will have to select where to do the mod operations. The result in Table 3.2 assumes the optimal positioning of remainder operations in Buffet.

To the best of our knowledge, Geppetto does not (yet) support mod operations when the modulus is not a power of two, so it's assumed that the programmer will have to manually add the additional inputs and constraint checking of the remainder operations, plus choosing where to perform the remainder operations.

RSA-2048 Modular Exponentiation. Due to the complexity of the RSA circuit, we only compare with existing implementations/specifications, such as the state-of-the-art implementation in Cinderella [9], which was developed on top of Geppetto. It is true that Buffet as well provides a library for long integer operations, however the remainder operation is not implemented, and to implement it efficiently, it would require the programmer to specify prover witness inputs to the circuit.

To ensure a fair comparison, we implemented the specification provided in the Cinderella paper (assuming a pre-known modulus), and compared it with our back end technique described earlier (The cost of our Cinderella implementation is less, which provides a good lower bound). Cinderella's implementation divides the big integers to 120-bit words, and hence cannot apply the group step described in our equality assertion algorithm, while in xJsnark's case, the back end sets the bitwidth to 32, applies our $O(n)$ multiplication algorithm, and the improved equality assertion algorithm with the additional group step. The result is shown in Table 3.2, showing more than **1.5**$\times$ speed-up **overall**. Looking closer, the enhancement in the

Table 3.2: Comparison between different compilers with respect to the number of constraints and programmability. A filled circle indicates more effort/experience by the programmer relatively. A † symbol indicates a conservative lower bound.

	Buffet [11]		Geppetto [12]		xJsnark	
SHA-256	44999	○	38556	○	26155	○
SWIFFT	3857	◐	3006†	●	3006	○
RSA-2048	-	●	144933 [9]	◐	90804	○
AES-128 (300)	9.3×10^{6}†	○	27.2×10^{6}†	◐	4.2×10^{6}	○

equality assertion step exceeds **3**×, as both implementations share about 70,000 constraints for verifying the range of prover witness values. Note that in our case, the programmer does not deal with any additional witness inputs or constraints, compared to Cinderella's code in Geppetto. This all happens in the background.

AES-128. The major cost incurred by an AES block in naive implementations is mainly due to the cost of randomly accessing its S-Box, therefore we focus in this section only on this part while assuming that the rest of the AES function has been implemented optimally for all the other compilers. This is in particular to show the savings that our proposed memory technique introduces. Table 3.2 illustrates the results, when the number of AES blocks is high, e.g. 300. (Note that our technique always provides better results (Figure 3.8)). To the best of our knowledge, Geppetto does not currently support random accesses of unknown indices, therefore the linear scan method is the default method to implement S-Box there. For the Buffet case, we computed an estimate using the equation provided in the original paper [11], which uses an unoptimized permutation network. As shown in the table, our $O(\sqrt{n})$ technique provides more than **2**× speedup over Buffet for the whole AES circuit. It should also be noted that our $O(\sqrt{n})$ approach used in this evaluation achieves

Table 3.3: Comparison between an improved permutation network approach based on Buffet [1] versus xJsnark's static read only memory technique, on a circuit of 300 AES-128 blocks

	Proof Time	Proving Key Size	Memory Usage
Buffet Permutation Network (improved)	347.32 s	1.8 GB	24 GB
xJsnark	172.62 s	958.79 MB	16 GB

1.7× enhancement over the **optimized** permutation network approach we use. The savings also apply to the key sizes and the memory usage.

Table 3.3 shows a comparison between the proposed memory approach, and an improved permutation network approach (based on the approach of Buffet [1]) in terms of the evaluation key size and the memory usage. Results were obtained by running key generation and a sample proof computation using libsnark [15]. The table illustrates better numbers for xJsnark's approach (2× better in both the proof time and the proving key size). This confirms that the savings obtained by reducing the number of multiplication gates is much more than the cost of adding constant multiplications in the circuit (the proof time is measured on an Amazon r3.8x EC2 instance).

3.6.2 Random Memory Access Application

In this section, we discuss the savings introduced by our framework in sorting applications. We start by comparing the result of compiling merge sorting code using Buffet [11], and xJsnark. The first two columns of Table 3.4compare the circuit sizes produced by a merge sort implementation of an array of 16-bit integers, that is

Table 3.4: Number of constraints for sorting circuits (n: input size)

n	Buffet [11] Merge Sort	xJsnark Merge Sort	xJsnark Verify Permut.
32	276×10^3	79×10^3	782
64	714×10^3	266×10^3	1646
512	7.9×10^6 [11]	3.8×10^6	14830

according to Buffet's available repository [61]. The code written using xJsnark is almost similar except for minor syntax differences. For the first case where $n = 32$, our adaptive memory algorithm selects the linear scan method over the permutation network after analyzing the memory workload in the first phase. In the other cases, the linear scan performs worse, and the back end selects the permutation network instead. In both cases, there is **2-3**$\times$ improvement over earlier implementations.

Furthermore, note that it will be more efficient to write code for verifying the sorting result directly, using the high-level permutation verification feature introduced in Section 3.2. Although the merge sort code uses also a permutation network in the background, the permutation verification method provides significantly better results as it saves a logarithmic factor of comparisons, and eliminates the cost of read/write memory operations. Table 3.4 shows the savings compared to basic approaches (Appendix C.1.1 provides a code example).

3.6.3 ZeroCash's ZK-SNARK Circuit

Using our framework, we developed one existing application that was manually developed using the libsnark gadget library [15, 62], mainly the pour circuit in the ZeroCash system [7], which is used to add privacy to transactions on top of the

Table 3.5: Comparison between the manual implementation and different compilers in the case of ZeroCash's Pour Circuit. A filled circle indicates more effort/experience by the programmer. A † symbol indicates a conservative lower bound.

	# Constraints	Development Effort
Existing Manual Implementations	4×10^6 [7, 62]	●
xJsnark	3.81×10^6	○
Buffet [11]	$6 \times 10^{6\dagger}$	○
Geppetto [12]	$5 \times 10^{6\dagger}$	○

blockchain.

Table 3.5 compares the alternatives for developing the ZeroCash Pour circuit. The reason xJsnark provides slightly better results than the manual optimized implementation is due to some further low-level arithmetic optimizations that can be automatically detected such as those by the multi-variate polynomial minimizer, by detecting similarities across loops (As described in Section 3.5). In terms of the development effort, the implementation is more compact in comparison with the existing available implementation online on Github [62], and the gadgets it uses from libsnark [15]. The rest of the table shows the efficiency achieved by xJsnark compared to other compilers.

3.7 Limitations and Future Work

In this section, we discuss the limitations of our current implementation, and directions for future work.

1. Integration of other optimizations. Previous implementations like Buffet or Geppetto have other orthogonal optimizations that we plan to integrate in our next

implementation. For example, Buffet provides a technique for loop coalescing, which helps to reduce the complexity of nested loops, when the total running time is $O(n)$, while the trivial compilation to SNARK circuits can lead to $O(n^2)$ size. This can be helpful for some applications, beyond what we evaluated. An optimization implemented by Geppetto is energy-saving circuits, which reduces the prover's running time by making all the wire values for not taken branches have a zero value. Other optimizations include: dead code elimination, which ignores any parts of the circuit that did not contribute to the output of the circuit. Most of such optimizations can be integrated in our back end. Another direction would be to formally argue about the correctness of the compiler as in the PinocchioQ compiler [63].

2. Front end alternatives. As illustrated earlier, we used Jetbrains MPS to build our Java extension for the front end. One possible drawback of using MPS Jetbrains is that in order for the programmers to develop the code, it has to be done in the projectional editor provided by MPS. Although this framework is free and can be used on top of Windows, Linux, OS X and others, we plan to make our implementation more generic, and investigate other approaches for developing the java extension in order for our framework to be more accessible. Note that the optimizations described in our back end does not depend on the specific framework of the front end, and can be integrated with any other front end providing a similar interface.

Chapter 4: HAWK: Privacy-preserving Smart Contracts [1]

Emerging smart contract systems, such as Ethereum [19], enable user-defined decentralized applications to run on top of a public blockchain system. However, the most commonly used form of these technologies lacks transactional privacy. All the inputs and outputs of a smart contract execution have to be publicly visible to the network in order to achieve transparency. Even though parties can create new pseudonymous public keys to increase their anonymity, the values of all transactions and balances for each (pseudonymous) public key are publicly visible. Furthermore, recent works have also demonstrated deanonymization attacks by analyzing the transactional graph structures of cryptocurrencies [64, 65].

We believe that lack of privacy could be a major hindrance towards the broad adoption of decentralized smart contracts, since financial transactions (e.g., insurance contracts or stock trading) are considered by many individuals and organizations as being highly secret. Although there has been progress in designing privacy-preserving cryptocurrencies such as Zerocash [66] and several others [6, 67], these systems forgo programmability, and it is unclear *a priori* how to enable programmability without exposing transactions and data in cleartext to miners.

[1] This chapter is based on a joint work with Andrew Miller, Elaine Shi, Zikai Wen and Charalampos Papamanthou, that appeared in the IEEE Symposium on Security and Privacy 2016 [2].

4.1 HAWK Overview

In the following, we consider the setting of a public blockchain system with support for smart contracts, e.g. Ethereum [19].

We propose HAWK, a framework for building privacy-preserving smart contracts. With HAWK, a *non-specialist* programmer can easily write a HAWK program without having to implement any cryptography. Our HAWK compiler is in charge of compiling the program to a cryptographic protocol between the blockchain and the users. A HAWK program contains two parts:

1. A *private* portion denoted ϕ_{priv} which takes in parties' input data (e.g., choices in a "rock, paper, scissors" game) as well as currency units (e.g., bids in an auction). ϕ_{priv} performs computation to determine the payout distribution amongst the parties. For example, in an auction, winner's bid goes to the seller, and others' bids are refunded. The private HAWK program ϕ_{priv} is meant to protect the participants' data and the exchange of money.

2. A *public* portion denoted ϕ_{pub} that does not touch private data or money.

Our compiler will compile the HAWK program into the following pieces which jointly define a cryptographic protocol between users, the manager, and the blockchain:

- the blockchain's program which will be executed by all consensus nodes;
- a program to be executed by the users; and
- a program to be executed by a special facilitating party called the manager

which will be explained shortly.

Security guarantees. HAWK's security guarantees encompass two aspects:

- *On-chain privacy.* On-chain privacy stipulates that transactional privacy be provided against the public (i.e., against any party *not* involved in the contract) – unless the contractual parties themselves voluntarily disclose information. Although in HAWK protocols, users exchange data with the blockchain, and rely on it to ensure fairness against aborts, the flow of money and amount transacted in the private HAWK program ϕ_{priv} is cryptographically hidden from the public's view. Informally, this is achieved by sending "encrypted" information to the blockchain, and relying on zero-knowledge proofs to enforce the correctness of contract execution and money conservation.

- *Contractual security.* While on-chain privacy protects contractual parties' privacy against the public (i.e., parties not involved in the financial contract), contractual security protects parties in the same contractual agreement from *each other*. HAWK assumes that contractual parties act *selfishly* to maximize their own financial interest. In particular, they can *arbitrarily* deviate from the prescribed protocol or even *abort* prematurely. Therefore, contractual security is a multi-faceted notion that encompasses not only cryptographic notions of confidentiality and authenticity, but also financial fairness in the presence of cheating and aborting behavior. The best way to understand contractual security is through a concrete example, and we refer the reader to Section 4.1.1 for a more detailed explanation.

Minimally trusted manager. The execution of HAWK contracts are facilitated by a special party called the manager. The manager can see the users' inputs and is trusted not to disclose users' private data. However, the manager is not trusted for the correctness of the computation. Even when the manager can deviate arbitrarily from the protocol or collude with the parties, the manager cannot affect the correct execution of the contract. In the event that a manager aborts the protocol, it can be financially penalized, and users obtain compensation accordingly.

The manager also need not be trusted to maintain the security or privacy of the underlying currency (e.g., it cannot double-spend, inflate the currency, or deanonymize users). Furthermore, if multiple contract instances run concurrently, each contract may specify a different manager and the effects of a corrupt manager are confined to that instance. Finally, the manager role may be instantiated with trusted computing hardware like Intel SGX, or replaced with a multiparty computation among the users themselves, as we describe in Section 4.3.3.

4.1.1 Example: Sealed Auction

Example program. Figure 4.1 shows a HAWK program for implementing a sealed, second-price auction where the highest bidder wins, but pays the second highest price. Second-price auctions are known to incentivize truthful bidding under certain assumptions, [68] and it is important that bidders submit bids without knowing the bid of the other people. Our example auction program contains a private portion

ϕ_{priv} that determines the winning bidder and the price to be paid; and a public portion ϕ_{pub} that relies on public deposits to protect bidders from an aborting manager.

For the time being, we assume that the set of bidders are known *a priori*.

Contractual security requirements. HAWK will compile this auction program to a cryptographic protocol. As mentioned earlier, as long as the bidders and the manager do not voluntarily disclose information, transaction privacy is maintained against the public. HAWK also guarantees the following contractual security requirements for parties in the contract:

- *Input independent privacy.* Each user does not see others' bids before committing to their own (even when they collude with a potentially malicious manager). This way, users bids are independent of others' bids.

- *Posterior privacy.* As long as the manager does not disclose information, users' bids are kept private from each other (and from the public) even after the auction.

- *Financial fairness.* Parties may attempt to prematurely abort from the protocol to avoid payment or affect the redistribution of wealth. If a party aborts or the auction manager aborts, the aborting party will be financially penalized while the remaining parties receive compensation. As is well-known in the cryptography literature, such fairness guarantees are not attainable in general by off-chain only protocols such as secure multi-party computation [29, 30]. As explained later, HAWK offers built-in mechanisms for enforcing refunds of

private bids after certain timeouts.

- *Security against a dishonest manager.* We ensure *authenticity* against a dishonest manager: besides aborting, a dishonest manager cannot affect the outcome of the auction and the redistribution of money, even when it colludes with a subset of the users. To ensure the above, input independent privacy against a faulty manager is a prerequisite. Moreover, if the manager aborts, it can be financially penalized, and the participants obtain corresponding remuneration.

An auction with the above security and privacy requirements cannot be trivially implemented atop existing cryptocurrency systems such as Ethereum [19] or Zerocash [66]. The former allows for programmability but does not guarantee transactional privacy, while the latter guarantees transactional privacy but at the price of even reduced programmability than Bitcoin.

Aborting and timeouts. Aborting is dealt with using timeouts. A HAWK program such as Figure 4.1 declares timeout parameters. Three timeouts are declared where $T_1 < T_2 < T_3$:

T_1 : The HAWK contract stops collecting bids after T_1.

T_2 : All users should have opened their bids to the manager within T_2; if a user submitted a bid but fails to open by T_2, its input bid is treated as 0 (and any other potential input data treated as $\perp$), such that the manager can continue.

T_3 : If the manager aborts, users can reclaim their private bids after time T_3.

The public HAWK contract ϕ_{pub} can additionally implement incentive structures. Our sealed auction program redistributes the manager's public deposit if it aborts. Specifically, in our sealed auction program, ϕ_{pub} defines two functions, namely `check` and `managerTimeOut`. The `check` function will be invoked when the HAWK contract completes execution within T_3, i.e., manager did not abort. Otherwise, if the HAWK contract does not complete execution within T_3, the `managerTimeOut` function will be invoked. We remark that although not explicitly written in the code, all HAWK contracts have an implicit default entry point for accepting parties' deposits – these deposits are withheld by the contract till they are redistributed by the contract. Bidders should check that the manager has made a public deposit before submitting their bids.

Additional applications. Besides the sealed auction example, We illustrate that HAWK can support various other applications. We give more sample programs in Section 4.5.2.

4.1.2 Contributions

To the best of our knowledge, HAWK was the first to simultaneously offer transactional privacy and programmability in a decentralized cryptocurrency system.

New cryptography suite. We implement a new cryptography suite that binds private transactions with programmable logic. Our protocol suite contains three essential primitives `freeze`, `compute`, and `finalize`. The `freeze` primitive allows

parties to commit to not only normal data, but also coins. Committed coins are frozen in the contract, and the payout distribution will later be determined by the program ϕ_{priv}. During `compute`, parties open their committed data and currency to the manager, such that the manager can compute the function ϕ_{priv}. Based on the outcome of ϕ_{priv}, the manager now constructs new private coins to be paid to each recipient. The manager then submits to the blockchain both the new private coins as well as zero-knowledge proofs of their well-formedness. At this moment, the previously frozen coins are now redistributed among the users. Our protocol suite strictly generalizes Zerocash since Zerocash implements only private money transfers between users without programmability.

Implementation and evaluation. We built a HAWK prototype and evaluated its performance by implementing several example applications, including a *sealed-bid auction*, a *"rock, paper, scissors"* game, a *crowdfunding* application, and a *swap financial instrument*. We propose interesting protocol optimizations that gained us a factor of **10**× in performance relative to a straightforward implementation. We show that for at about 100 parties (e.g., auction and crowdfunding), the manager's cryptographic computation (the most expensive part of the protocol) is under **2.85min using 4 cores**, translating to under **$0.14** of EC2 time. Further, all on-chain computation (performed by all miners) is very cheap, and under **20**ms for all cases. We will open source our HAWK framework in the near future.

```
1  HawkDeclareParties(Seller,/* N parties */);
2  HawkDeclareTimeouts(/* hardcoded timeouts */);

3    // Private portion φ_priv
4  private contract auction(Inp &in, Outp &out) {
5    int winner = -1;
6    int bestprice = -1;
7    int secondprice = -1;

8    for (int i = 0; i < N; i++) {
9      if (in.party[i].$val > bestprice) {
10       secondprice = bestprice;
11       bestprice = in.party[i].$val;
12       winner = i;
13     } else if (in.party[i].$val > secondprice) {
14       secondprice = in.party[i].$val;
15     }
16   }

17   // Winner pays secondprice to seller
18   // Everyone else is refunded
19   out.Seller.$val = secondprice;
20   out.party[winner].$val = bestprice-secondprice;
21   out.winner = winner;
22   for (int i = 0; i < N; i++) {
23     if (i != winner)
24       out.party[i].$val = in.party[i].$val;
25   }
26 }

27   // Public portion φ_pub
28 public contract deposit {
29   // Manager deposited $N earlier
30   def check():   // invoked on contract completion
31     send $N to Manager  // refund manager
32   def managerTimeOut():
33     for (i in range($N)):
34       send $1 to party[i]
35 }
```

Figure 4.1: HAWK program for a second-price sealed auction. The provided code is an approximation of our real implementation.

4.2 Notations and Threat Model

We begin by describing the trust model and assumptions.

In this work, the blockchain refers to a decentralized set of miners who run a secure consensus protocol to agree upon the global state. We therefore will regard the blockchain as a conceptual trusted party who is **trusted for correctness and availability, but not trusted for privacy**. The blockchain not only maintains a global ledger that stores the balance for every pseudonym, but also executes user-defined programs. More specifically, we make the following assumptions:

- *Time.* The blockchain is aware of a discrete clock that increments in *rounds*.
- *Public state.* All parties can observe the state of the blockchain. This means that all parties can observe the public ledger on the blockchain, as well as the state of any user-defined blockchain program (part of a contract protocol).
- *Message delivery.* Messages sent to the blockchain will arrive at the beginning of the next round. A network adversary may arbitrarily reorder messages that are sent to the blockchain within the same round. This means that the adversary may attempt a front-running attack (also referred to as the rushing adversary by cryptographers), e.g., upon observing that an honest user is trading a stock, the adversary preempts by sending a race transaction trading the same stock. Our protocols should be secure despite such adversarial message delivery schedules.

We assume that all parties have a reliable channel to the blockchain, and the

adversary cannot drop messages a party sends to the blockchain. In reality, this means that the overlay network must have sufficient redundancy. However, an adversary *can* drop messages delivered between parties off the blockchain.

- *Pseudonyms.* Users can make up an unbounded polynomial number of pseudonyms when communicating with the blockchain.

- *Correctness and availability.* We assume that the blockchain will perform any prescribed computation correctly. We also assume that the blockchain is always available.

4.2.1 Notations

Pseudonymity. All party identifiers that appear in our protocols by default refer to *pseudonyms*. When we write "upon receiving message from *some* P", this accepts a message from any pseudonym. Whenever we write "upon receiving message from P", without the keyword *some*, this accepts a message from a fixed pseudonym P, and typically which pseudonym we refer to is clear from the context.

Whenever we write "send m to $\mathcal{G}(\mathsf{Contract})$ as nym P" inside a user program, we assume that authentication is happening in the background (This is formally described in the Blockchain model in the full version of the work [2]) When the context is clear, we avoid writing "as nym P", and simply write "send m to $\mathcal{G}(\mathsf{Contract})$".

Ledger and money transfers. A public ledger is denoted ledger in our constructions. When a party sends $\$\mathsf{amt}$ to a blockchain program, this represents an ordinary

message transmission. Money transfers only take place when the blockchain programs update the public ledger ledger. In other words, the symbol $ is only adopted for readability (to distinguish variables associated with money and other variables), and does not have special meaning or significance. One can simply think of this variable as having the money type.

4.3 Cryptographic Protocols

In this section, we provide the cryptographic protocols we used for HAWK. Our protocols are broken down into two parts: 1) the private cash part that implements direct money transfers between users; and 2) the HAWK-specific part that binds transactional privacy with programmable logic. The formal protocol descriptions are given in Figures 4.2, 4.3, 4.4, 4.6 and 4.7. The ideal functionalities and the proofs are provided in the full version of the paper [2]. Below we explain the high-level intuition.

4.3.1 Warmup: Private Cash and Money Transfers

Our construction adopts a Zerocash-like protocol for implementing private cash and private currency transfers. For completeness, we give a brief explanation below, and we mainly focus on the `pour` operation which is technically more interesting. The blockchain program $\mathsf{Blockchain}_{\mathrm{cash}}$ maintains a set Coins of private coins. Each private coin is of the format

$$(\mathcal{P}, \mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val}))$$

where $\mathcal{P}$ denotes a party's pseudonym, and coin commits to the coin's value $\$\mathsf{val}$ under randomness s.

During a `pour` operation, the spender $\mathcal{P}$ chooses two coins in Coins to spend, denoted $(\mathcal{P}, \mathsf{coin}_1)$ and $(\mathcal{P}, \mathsf{coin}_2)$ where $\mathsf{coin}_i := \mathsf{Comm}_{s_i}(\$\mathsf{val}_i)$ for $i \in \{1, 2\}$. The `pour` operation pays val'_1 and val'_2 amount to two output pseudonyms denoted $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively, such that $\mathsf{val}_1 + \mathsf{val}_2 = \mathsf{val}'_1 + \mathsf{val}'_2$. The spender chooses new randomness s'_i for $i \in \{1, 2\}$, and computes the output coins as

$$\left(\mathcal{P}_i, \mathsf{coin}_i := \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)\right)$$

The spender gives the values s'_i and val'_i to the recipient $\mathcal{P}_i$ for $\mathcal{P}_i$ to be able to spend the coins later.

Now, the spender computes a zero-knowledge proof to show that the output coins are constructed appropriately, where correctness compasses the following aspects:

- *Existence of coins being spent.* The coins being spent $(\mathcal{P}, \mathsf{coin}_1)$ and $(\mathcal{P}, \mathsf{coin}_2)$ are indeed part of the private pool Coins. We remark that here the zero-knowledge property allows the spender to hide which coins it is spending – this is the key idea behind transactional privacy.

  To prove this efficiently, $\mathsf{Blockchain}_{\mathrm{cash}}$ maintains a Merkle tree MT over the private pool Coins. Membership in the set can be demonstrated by a Merkle branch consistent with the root hash, and this is done in zero-knowledge.

- *No double spending.* Each coin $(\mathcal{P}, \mathsf{coin})$ has a cryptographically unique serial number sn that can be computed as a pseudorandom function of $\mathcal{P}$'s secret

key and coin. To pour a coin, its serial number sn must be disclosed, and a zero-knowledge proof given to show the correctness of sn. $\mathsf{Blockchain}_{\mathrm{cash}}$ checks that no sn is used twice.

- *Money conservation.* The zero-knowledge proof also attests to the fact that the input coins and the output coins have equal total value.

We make some remarks about the security of the scheme. Intuitively, when an honest party pours to an honest party, the adversary $\mathcal{A}$ does not learn the values of the output coins assuming that the commitment scheme Comm is hiding, and the NIZK scheme we employ is computational zero-knowledge. The adversary $\mathcal{A}$ can observe the nyms that receive the two output coins. However, as we remarked earlier, since these nyms can be one-time, leaking them to the adversary would be okay. Essentially we only need to break linkability at spend time to ensure transactional privacy.

When a corrupted party $\mathcal{P}^*$ pours to an honest party $\mathcal{P}$, even though the adversary knows the opening of the coin, it cannot spend the coin $(\mathcal{P}, \mathsf{coin})$ once the transaction takes effect by the $\mathsf{Blockchain}_{\mathrm{cash}}$, since $\mathcal{P}^*$ cannot demonstrate knowledge of $\mathcal{P}$'s secret key. Since the contract binds the owner's nym $\mathcal{P}$ to the coin, only the owner can spend it even when the opening of coin is disclosed.

$\mathsf{Blockchain}_{\mathrm{cash}}$

Init: crs: a reference string for the underlying NIZK system

Coins: a set of coin commitments, initially $\emptyset$

$\mathsf{SpentCoins}$: set of spent serial numbers, initially $\emptyset$

Mint: Upon receiving $(\texttt{mint}, \$\mathsf{val}, s)$ from some party $\mathcal{P}$,

$\mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val})$

assert $(\mathcal{P}, \mathsf{coin}) \notin \mathsf{Coins}$

assert $\mathsf{ledger}[\mathcal{P}] \geq \$\mathsf{val}$

$\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] - \$\mathsf{val}$

add $(\mathcal{P}, \mathsf{coin})$ to Coins

Pour: Anonymous receive $(\texttt{pour}, \pi, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}_i, \mathsf{ct}_i\}_{i\in\{1,2\}})$

let MT be a merkle tree built over Coins

$\mathsf{statement} := (\mathsf{MT.root}, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}_i\}_{i\in\{1,2\}})$

assert $\mathsf{NIZK.Verify}(\mathcal{L}_{\texttt{POUR}}, \pi, \mathsf{statement})$

for $i \in \{1, 2\}$,

  assert $\mathsf{sn}_i \notin \mathsf{SpentCoins}$

  assert $(\mathcal{P}_i, \mathsf{coin}_i) \notin \mathsf{Coins}$

  add sn_i to $\mathsf{SpentCoins}$

  add $(\mathcal{P}_i, \mathsf{coin}_i)$ to Coins

  send $(\texttt{pour}, \mathsf{coin}_i, \mathsf{ct}_i)$ to $\mathcal{P}_i$,

---

Relation $(\mathsf{statement}, \mathsf{witness}) \in \mathcal{L}_{\texttt{POUR}}$ is defined as:

parse $\mathsf{statement}$ as $(\mathsf{MT.root}, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}'_i\}_{i\in\{1,2\}})$

parse $\mathsf{witness}$ as $(\mathcal{P}, \mathsf{sk}_{\mathtt{prf}}, \{\mathsf{branch}_i, s_i, \$\mathsf{val}_i, s'_i, r_i, \$\mathsf{val}'_i\})$

assert $\mathcal{P}.\mathsf{pk}_{\mathtt{prf}} = \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(0)$

assert $\$\mathsf{val}_1 + \$\mathsf{val}_2 = \$\mathsf{val}'_1 + \$\mathsf{val}'_2$

for $i \in \{1, 2\}$,

  $\mathsf{coin}_i := \mathsf{Comm}_{s_i}(\$\mathsf{val}_i)$

  assert $\mathsf{MerkleBranch}(\mathsf{MT.root}, \mathsf{branch}_i, (\mathcal{P} \| \mathsf{coin}_i))$

  assert $\mathsf{sn}_i = \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(\mathcal{P} \| \mathsf{coin}_i)$

  assert $\mathsf{coin}'_i = \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)$

Figure 4.2: $\mathsf{Blockchain}_{\mathrm{cash}}$ construction. A trusted setup phase generates the NIZK's common reference string crs. For notational convenience, we omit writing the crs explicitly in the construction. The Merkle tree MT is stored on the blockchain and not computed on the fly – we omit stating this in the protocol for notational simplicity.

Protocol $\mathsf{UserP}_{\mathbf{cash}}$

Init: Wallet: stores $\mathcal{P}$'s spendable coins, initially $\emptyset$

GenNym: sample a random seed $\mathsf{sk}_{\mathtt{prf}}$
$\mathsf{pk}_{\mathtt{prf}} := \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(0)$
return $\mathsf{pk}_{\mathtt{prf}}$

Mint: On input $(\mathtt{mint}, \$\mathsf{val})$,
sample a commitment randomness s
$\mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val})$
store $(s, \$\mathsf{val}, \mathsf{coin})$ in Wallet
send $(\mathtt{mint}, \$\mathsf{val}, s)$ to $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{cash}})$

Pour (as sender): On input $(\mathtt{pour}, \$\mathsf{val}_1, \$\mathsf{val}_2, \mathcal{P}_1, \mathcal{P}_2, \$\mathsf{val}'_1, \$\mathsf{val}'_2)$,
assert $\$\mathsf{val}_1 + \$\mathsf{val}_2 = \$\mathsf{val}'_1 + \$\mathsf{val}'_2$
for $i \in \{1,2\}$, assert $(s_i, \$\mathsf{val}_i, \mathsf{coin}_i) \in$ Wallet for some (s_i, coin_i)
let MT be a merkle tree over $\mathsf{Blockchain}_{\mathrm{cash}}.\mathsf{Coins}$
for $i \in \{1,2\}$:
  remove one $(s_i, \$\mathsf{val}_i, \mathsf{coin}_i)$ from Wallet
  $\mathsf{sn}_i := \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(\mathcal{P} \| \mathsf{coin}_i)$
  let branch_i be the branch of $(\mathcal{P}, \mathsf{coin}_i)$ in MT
  sample randomness s'_i, r_i
  $\mathsf{coin}'_i := \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)$
  $\mathsf{ct}_i := \mathsf{ENC}(\mathcal{P}_i.\mathsf{epk}, r_i, \$\mathsf{val}'_i \| s'_i)$
$\mathsf{statement} := (\mathsf{MT.root}, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}'_i\}_{i\in\{1,2\}})$
$\mathsf{witness} := (\mathcal{P}, \mathsf{sk}_{\mathtt{prf}}, \{\mathsf{branch}_i, s_i, \$\mathsf{val}_i, s'_i, r_i, \$\mathsf{val}'_i\})$
$\pi := \mathsf{NIZK.Prove}(\mathcal{L}_{\mathtt{POUR}}, \mathsf{statement}, \mathsf{witness})$
$\mathsf{AnonSend}(\mathtt{pour}, \pi, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in\{1,2\}})$
to $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{cash}})$

Pour (as recipient): On receive $(\mathtt{pour}, \mathsf{coin}, \mathsf{ct})$ from $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{cash}})$:
let $(\$\mathsf{val} \| s) := \mathsf{DEC}(\mathsf{esk}, \mathsf{ct})$
assert $\mathsf{Comm}_s(\$\mathsf{val}) = \mathsf{coin}$
store $(s, \$\mathsf{val}, \mathsf{coin})$ in Wallet
output $(\mathtt{pour}, \$\mathsf{val})$

Figure 4.3: $\mathsf{UserP}_{\mathrm{cash}}$ construction. Note that $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{cash}})$ is a functionality wrapper for $\mathsf{Blockchain}_{\mathrm{cash}}$ that captures the blockchain properties and threat model [2].

4.3.2 Binding Privacy and Programmable Logic

So far, $\mathsf{Blockchain}_{\mathrm{cash}}$, similar to Zerocash [66], only supports *direct* money transfers between users. We allow transactional privacy and programmable logic simutaneously.

Freeze. We support a new operation called `freeze`, that does not spend directly to a user, but commits the money as well as an accompanying private input to a smart contract. This is done using a `pour`-like protocol:

- The user $\mathcal{P}$ chooses a private coin $(\mathcal{P}, \mathsf{coin}) \in \mathsf{Coins}$, where $\mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val})$. Using its secret key, $\mathcal{P}$ computes the serial number sn for coin – to be disclosed with the `freeze` operation to prevent double-spending.

- The user $\mathcal{P}$ computes a commitment $(\mathsf{val}||\mathsf{in}||k)$ to the contract where in denotes its input, and k is a symmetric encryption key that is introduced due to a practical optimization explained later in Section 4.4.

- The user $\mathcal{P}$ now makes a zero-knowledge proof attesting to similar statements as in a `pour` operation, i.e., that the spent coin exists in the pool Coins, the sn is correctly constructed, and that the val committed to the contract equals the value of the coin being spent. See $\mathcal{L}_{\texttt{FREEZE}}$ in Figure 4.5 for details of the NP statement being proven.

Compute. Next, computation takes place off-chain to compute the payout distribution $\{\mathsf{val}'_i\}_{i\in[n]}$ and a proof of correctness. In HAWK, we rely on a minimally

trusted manager $\mathcal{P}_{\mathcal{M}}$ to perform computation. All parties would open their inputs to the manager $\mathcal{P}_{\mathcal{M}}$, and this is done by encrypting the opening to the manager's public key:

$$\mathsf{ct} := \mathsf{ENC}(\mathcal{P}_{\mathcal{M}}.\mathsf{epk}, r, (\$\mathsf{val} \| \mathsf{in} \| k \| s'))$$

The ciphertext ct is submitted to the smart contract along with appropriate zero-knowledge proofs of correctness. While the user can also directly send the opening to the manager off-chain, passing the ciphertext ct through the smart contract would make any aborts evident such that the contract can financially punish an aborting user.

After obtaining the openings, the manager now computes the payout distribution $\{\mathsf{val}'_i\}_{i\in[n]}$ and public output out by applying the private contract ϕ_{priv}. The manager also constructs a zero-knowledge proof attesting to the outcomes.

Finalize. When the manager submits the outcome of ϕ_{priv} and a zero-knowledge proof of correctness to $\mathsf{Blockchain}_{\text{hawk}}$, $\mathsf{Blockchain}_{\text{hawk}}$ verifies the proof and redistributes the frozen money accordingly. Here $\mathsf{Blockchain}_{\text{hawk}}$ also passes the manager's public input $\mathsf{in}_{\mathcal{M}}$ and public output out to the public HAWK contract ϕ_{pub}. The public contract ϕ_{pub} can be invoked to check the validity of the manager's input, as well as redistribute public collateral deposit.

Assuming that the hash function in the Merkle tree is collision resistant, the commitment scheme Comm is perfectly binding and computationally hiding, the NIZK scheme is computationally zero-knowledge and simulation sound extractable, the encryption schemes ENC and SENC are perfectly correct and semantically se-

cure, the PRF scheme PRF is secure, then it can be shown that our protocols in Figures 4.2, 4.3, 4.6 and 4.7 are secure (The theorem and proofs are available in the full version of the paper [2]).

$\mathsf{Blockchain}_{\mathrm{hawk}}(\mathcal{P}_{\mathcal{M}}, \{\mathcal{P}_i\}_{i\in[N]}, T_1, T_2, \phi_{\mathrm{priv}}, \phi_{\mathrm{pub}})$

Init: Call $\mathsf{Blockchain}_{\mathrm{cash}}$.**Init**.

Freeze: Upon receiving $(\texttt{freeze}, \pi, \mathsf{sn}_i, \mathsf{cm}_i)$ from $\mathcal{P}_i$:

- assert current time $T \leq T_1$
- assert this is the first `freeze` from $\mathcal{P}_i$
- let MT be a merkle tree built over Coins
- assert $\mathsf{sn}_i \notin \mathsf{SpentCoins}$
- $\mathsf{statement} := (\mathcal{P}_i, \mathsf{MT.root}, \mathsf{sn}_i, \mathsf{cm}_i)$
- assert $\mathsf{NIZK.Verify}(\mathcal{L}_{\texttt{FREEZE}}, \pi, \mathsf{statement})$
- add sn_i to $\mathsf{SpentCoins}$ and store cm_i for later

Compute: Upon receiving $(\texttt{compute}, \pi, \mathsf{ct})$ from $\mathcal{P}_i$:

- assert $T_1 \leq T < T_2$ for current time T
- assert $\mathsf{NIZK.Verify}(\mathcal{L}_{\texttt{COMPUTE}}, \pi, (\mathcal{P}_{\mathcal{M}}, \mathsf{cm}_i, \mathsf{ct}))$
- send $(\texttt{compute}, \mathcal{P}_i, \mathsf{ct})$ to $\mathcal{P}_{\mathcal{M}}$

Finalize: On receiving $(\texttt{finalize}, \pi, \mathsf{in}_{\mathcal{M}}, \mathsf{out}, \{\mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in[N]})$ from $\mathcal{P}_{\mathcal{M}}$:

- assert current time $T \geq T_2$
- for every $\mathcal{P}_i$ that has not called `compute`, set $\mathsf{cm}_i := \perp$
- $\mathsf{statement} := (\mathsf{in}_{\mathcal{M}}, \mathsf{out}, \{\mathsf{cm}_i, \mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in[N]})$
- assert $\mathsf{NIZK.Verify}(\mathcal{L}_{\texttt{FINALIZE}}, \pi, \mathsf{statement})$
- for $i \in [N]$:
  - assert $\mathsf{coin}'_i \notin \mathsf{Coins}$
  - add coin'_i to Coins
  - send $(\texttt{finalize}, \mathsf{coin}'_i, \mathsf{ct}_i)$ to $\mathcal{P}_i$
- Call ϕ_{pub}.check$(\mathsf{in}_{\mathcal{M}}, \mathsf{out})$

$\mathsf{Blockchain}_{\mathrm{cash}}$: include $\mathsf{Blockchain}_{\mathrm{cash}}$

ϕ_{pub} : include user-defined public contract ϕ_{pub}

Figure 4.4: $\mathsf{Blockchain}_{\mathrm{hawk}}$ construction.

Relation $(\mathsf{statement}, \mathsf{witness}) \in \mathcal{L}_{\texttt{FREEZE}}$ is defined as:

parse $\mathsf{statement}$ as $(\mathcal{P}, \mathsf{MT.root}, \mathsf{sn}, \mathsf{cm})$

parse $\mathsf{witness}$ as $(\mathsf{coin}, \mathsf{sk}_{\texttt{prf}}, \mathsf{branch}, s, \$\mathsf{val}, \mathsf{in}, k, s')$

$\mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val})$

assert $\mathsf{MerkleBranch}(\mathsf{MT.root}, \mathsf{branch}, (\mathcal{P} \| \mathsf{coin}))$

assert $\mathcal{P}.\mathsf{pk}_{\texttt{prf}} = \mathsf{sk}_{\texttt{prf}}(0)$

assert $\mathsf{sn} = \mathsf{PRF}_{\mathsf{sk}_{\texttt{prf}}}(\mathcal{P} \| \mathsf{coin})$

assert $\mathsf{cm} = \mathsf{Comm}_{s'}(\$\mathsf{val} \| \mathsf{in} \| k)$

Relation $(\mathsf{statement}, \mathsf{witness}) \in \mathcal{L}_{\texttt{COMPUTE}}$ is defined as:

parse $\mathsf{statement}$ as $(\mathcal{P}_{\mathcal{M}}, \mathsf{cm}, \mathsf{ct})$

parse $\mathsf{witness}$ as $(\$\mathsf{val}, \mathsf{in}, k, s', r)$

assert $\mathsf{cm} = \mathsf{Comm}_{s'}(\$\mathsf{val} \| \mathsf{in} \| k)$

assert $\mathsf{ct} = \mathsf{ENC}(\mathcal{P}_{\mathcal{M}}.\mathsf{epk}, r, (\$\mathsf{val} \| \mathsf{in} \| k \| s'))$

Relation $(\mathsf{statement}, \mathsf{witness}) \in \mathcal{L}_{\texttt{FINALIZE}}$ is defined as:

parse $\mathsf{statement}$ as $(\mathsf{in}_{\mathcal{M}}, \mathsf{out}, \{\mathsf{cm}_i, \mathsf{coin}'_i, \mathsf{ct}_i\}_{i \in [N]})$

parse $\mathsf{witness}$ as $\{s_i, \$\mathsf{val}_i, \mathsf{in}_i, s'_i, k_i\}_{i \in [N]}$

$(\{\$\mathsf{val}'_i\}_{i \in [N]}, \mathsf{out}) := \phi_{\text{priv}}(\{\$\mathsf{val}_i, \mathsf{in}_i\}_{i \in [N]}, \mathsf{in}_{\mathcal{M}})$

assert $\sum_{i \in [N]} \$\mathsf{val}_i = \sum_{i \in [N]} \$\mathsf{val}'_i$

for $i \in [N]$:

assert $\mathsf{cm}_i = \mathsf{Comm}_{s_i}(\$\mathsf{val}_i \| \mathsf{in}_i \| k_i))$
$\vee (\$\mathsf{val}_i, \mathsf{in}_i, k_i, s_i, \mathsf{cm}_i) = (0, \bot, \bot, \bot, \bot)$

assert $\mathsf{ct}_i = \mathsf{SENC}_{k_i}(s'_i \| \$\mathsf{val}'_i)$

assert $\mathsf{coin}'_i = \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)$

Figure 4.5: Relation definitions for the $\mathsf{Blockchain}_{\text{hawk}}$ and $\mathsf{UserP}_{\text{hawk}}$ constructions in Figures 4.4, 4.6 and 4.7

Protocol $\mathsf{UserP}_{\mathbf{hawk}}(\mathcal{P}_{\mathcal{M}}, \{\mathcal{P}_i\}_{i\in[N]}, T_1, T_2, \phi_{\mathbf{priv}}, \phi_{\mathbf{pub}})$

Init: Call $\mathsf{UserP}_{\mathrm{cash}}$.**Init**.

Protocol for a party $\mathcal{P} \in \{\mathcal{P}_i\}_{i\in[N]}$:

Freeze: On input $(\texttt{freeze}, \$\mathsf{val}, \mathsf{in})$ as party $\mathcal{P}$:

- assert current time $T < T_1$
- assert this is the first `freeze` input
- let MT be a merkle tree over $\mathsf{Blockchain}_{\mathrm{cash}}.\mathsf{Coins}$
- assert that some entry $(s, \$\mathsf{val}, \mathsf{coin}) \in \mathsf{Wallet}$ for some (s, coin)
- remove one $(s, \$\mathsf{val}, \mathsf{coin})$ from Wallet
- $\mathsf{sn} := \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(\mathcal{P}\|\mathsf{coin})$
- let branch be the branch of $(\mathcal{P}, \mathsf{coin})$ in MT
- sample a symmetric encryption key k
- sample a commitment randomness s'
- $\mathsf{cm} := \mathsf{Comm}_{s'}(\$\mathsf{val}\|\mathsf{in}\|k)$
- $\mathsf{statement} := (\mathcal{P}, \mathsf{MT.root}, \mathsf{sn}, \mathsf{cm})$
- $\mathsf{witness} := (\mathsf{coin}, \mathsf{sk}_{\mathtt{prf}}, \mathsf{branch}, s, \$\mathsf{val}, \mathsf{in}, k, s')$
- $\pi := \mathsf{NIZK.Prove}(\mathcal{L}_{\mathtt{FREEZE}}, \mathsf{statement}, \mathsf{witness})$
- send $(\texttt{freeze}, \pi, \mathsf{sn}, \mathsf{cm})$ to $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{hawk}})$
- store in, cm, $\$\mathsf{val}$, s', and k to use later (in `compute`)

Compute: On input $(\texttt{compute})$ as party $\mathcal{P}$:

- assert current time $T_1 \le T < T_2$
- sample encryption randomness r
- $\mathsf{ct} := \mathsf{ENC}(\mathcal{P}_{\mathcal{M}}.\mathsf{epk}, r, (\$\mathsf{val}\|\mathsf{in}\|k\|s'))$
- $\pi := \mathsf{NIZK.Prove}((\mathcal{P}_{\mathcal{M}}, \mathsf{cm}, \mathsf{ct}), (\$\mathsf{val}, \mathsf{in}, k, s', r))$
- send $(\texttt{compute}, \pi, \mathsf{ct})$ to $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{hawk}})$

Finalize: Receive $(\texttt{finalize}, \mathsf{coin}, \mathsf{ct})$ from $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{hawk}})$:

- decrypt $(s\|\$\mathsf{val}) := \mathsf{SDEC}_k(\mathsf{ct})$
- store $(s, \$\mathsf{val}, \mathsf{coin})$ in Wallet
- output $(\texttt{finalize}, \$\mathsf{val})$

$\mathsf{UserP}_{\mathrm{cash}}$: include $\mathsf{UserP}_{\mathrm{cash}}$.

Figure 4.6: $\mathsf{UserP}_{\mathrm{hawk}}$ construction: The participant protocol. $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{hawk}})$ is a functionality wrapper for $\mathsf{Blockchain}_{\mathrm{hawk}}$, that captures the blockchain properties and threat model [2].

Protocol $\mathsf{UserP}_{\mathbf{hawk}}(\mathcal{P}_{\mathcal{M}}, \{\mathcal{P}_i\}_{i\in[N]}, T_1, T_2, \phi_{\mathbf{priv}}, \phi_{\mathbf{pub}})$

Init: Call $\mathsf{UserP}_{\mathrm{cash}}$.**Init**.

Protocol for manager $\mathcal{P}_{\mathcal{M}}$:

Compute: On receive $(\texttt{compute}, \mathcal{P}_i, \mathsf{ct})$ from $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{hawk}})$:

decrypt and store $(\$\mathsf{val}_i \| \mathsf{in}_i \| k_i \| s_i) := \mathsf{DEC}(\mathsf{esk}, \mathsf{ct})$

store $\mathsf{cm}_i := \mathsf{Comm}_{s_i}(\$\mathsf{val}_i \| \mathsf{in}_i \| k_i)$

output $(\mathcal{P}_i, \$\mathsf{val}_i, \mathsf{in}_i)$

If this is the last `compute` received:

for $i \in [N]$ such that $\mathcal{P}_i$ has not called `compute`,
$(\$\mathsf{val}_i, \mathsf{in}_i, k_i, s_i, \mathsf{cm}_i) := (0, \bot, \bot, \bot, \bot)$

$(\{\$\mathsf{val}'_i\}_{i\in[N]}, \mathsf{out}) := \phi_{\mathrm{priv}}(\{\$\mathsf{val}_i, \mathsf{in}_i\}_{i\in[N]}, \mathsf{in}_{\mathcal{M}})$

store and output $(\{\$\mathsf{val}'_i\}_{i\in[N]}, \mathsf{out})$

Finalize: On input $(\texttt{finalize}, \mathsf{in}_{\mathcal{M}}, \mathsf{out})$:

assert current time $T \geq T_2$

for $i \in [N]$:

sample a commitment randomness s'_i

$\mathsf{coin}'_i := \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)$

$\mathsf{ct}_i := \mathsf{SENC}_{k_i}(s'_i \| \$\mathsf{val}'_i)$

$\mathsf{statement} := (\mathsf{in}_{\mathcal{M}}, \mathsf{out}, \{\mathsf{cm}_i, \mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in[N]})$

$\mathsf{witness} := \{s_i, \$\mathsf{val}_i, \mathsf{in}_i, s'_i, k_i\}_{i\in[N]}$

$\pi := \mathsf{NIZK.Prove}(\mathsf{statement}, \mathsf{witness})$

send $(\texttt{finalize}, \pi, \mathsf{in}_{\mathcal{M}}, \mathsf{out}, \{\mathsf{coin}'_i, \mathsf{ct}_i\})$ to $\mathcal{G}(\mathsf{Blockchain}_{\mathrm{hawk}})$

$\mathsf{UserP}_{\mathrm{cash}}$: include $\mathsf{UserP}_{\mathrm{cash}}$.

Figure 4.7: $\mathsf{UserP}_{\mathrm{hawk}}$ construction: The manager protocol.

4.3.3 Extensions and Discussions

Refunding frozen coins to users. In our implementation, we extend our basic scheme to allow the users to reclaim their frozen money after a timeout $T_3 > T_2$. To achieve this, user $\mathcal{P}$ simply sends the contract a newly constructed coin $(\mathcal{P}, \mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val}))$ and proves in zero-knowledge that its value $\mathsf{val} is equal to that of the frozen coin. In this case, the user can identify the previously frozen coin in the clear, i.e., there is no need to compute a zero-knowledge proof of membership within the frozen pool as is needed in a `pour` transaction.

Instantiating the manager with trusted hardware. In some applications, it may be a good idea to instantiate the manager using trusted hardware such as the emerging Intel SGX. In this case, the off-chain computation can take place in a secret SGX enclave that is not visible to any untrusted software or users. Alternatively, in principle, the manager role can also be split into two or more parties that jointly run a secure computation protocol – although this approach is likely to incur higher overhead.

Note that our model is fundamentally different from placing full trust in any centralized node. Trusted hardware cannot serve as a replacement of the blockchain. Any off-chain only protocol that does not interact with the blockchain cannot offer financial fairness in the presence of aborts – even when trusted hardware is employed.

Furthermore, even the use of SGX does not obviate the need for our cryptographic protocol. If the SGX is trusted only by a subset of parties (e.g., just the

parties to a particular private contact), rather than globally, then those users can benefit from the efficiency of an SGX-managed private contract, while still utilizing the more widely trusted underlying currency.

Pouring anonymously to long-lived pseudonyms.

In our protocols, the `pour` operation discloses the recipient's pseudonyms.

This means that our protocols only retain full privacy if the recipient generates a fresh, new pseudonym every time. In comparison, Zerocash [66] provides an option of anonymously spending to a long-lived pseudonym (in other words, having `pour` not disclose recipients' pseudonyms to the public).

It would be straightforward to add this feature to HAWK as well (at the cost of a constant factor blowup in performance); however, in most applications (e.g., a payment made after receiving an invoice), the transfer is subsequent to some interaction between the recipient and sender.

Open enrollment of pseudonyms. In our current contracts, we assume that parties' pseudonyms are hardcoded and known a priori. We can easily relax this to allow open enrollment of any pseudonym that joins the contract (e.g., in an auction). Our implementation supports open enrollment. Due to SNARK's pre-processing, right now, each contract instance must declare an upper-bound on the number of participants. An enrollment fee can potentially be adopted to prevent a DoS attack where the attacker joins the contract with many pseudonyms thus preventing legitimate users from joining. How to choose the correct fee amount to achieve incentive compatibility is left as an open research challenge. The a priori

upper bound on the number of participants can be avoided if we adopt recursively composable SNARKs [12, 69].

4.4 Adopting SNARKs in UC Protocols and Practical Optimizations

4.4.1 Using SNARKs in UC Protocols

Succinct Non-interactive ARguments of Knowledge [4, 25, 70] provide succinct proofs for general computation tasks, and have been implemented by several systems [1, 4, 25]. We would like to use SNARKs to instantiate the NIZK proofs in our protocols — unfortunately, SNARK's security is too weak to be directly employed in Universal Composability protocols [18]. Specifically, SNARK's knowledge extractor is non-blackbox and cannot be used by the UC simulator to extract witnesses from statements sent by the adversary and environment — doing so would require that the extractor be aware of the environment's algorithm, which is inherently incompatible with UC security.

UC protocols often require the NIZKs to have simulation extractability. Although SNARKs do not satisfy simulation extractability, in [16] we show that it is possible to apply efficient SNARK-lifting transformations to construct simulation extractable proofs from SNARKs [71]. Our implementations thus adopt those efficient SNARK-lifting transformations.

4.4.2 Practical Considerations

Efficient SNARK circuits. A SNARK prover's performance is mainly determined by the number of multiplication gates in the algebraic circuit to be proven (Section 2.1.1). To achieve efficiency, we designed optimized circuits through two ways: 1) using cryptographic primitives that are SNARK-friendly, i.e. efficiently realizable as arithmetic circuits under a specific SNARK parametrization. 2) Building customized circuit generators to produce SNARK-friendly implementations instead of relying on compilers to translate higher level implementation (See Appendices A and B for a summary of our tools).

The main cryptographic building blocks in our system are: collision-resistant hash function for the Merkle trees, pseudo-random function, commitment, and encryption. Our implementation supports both 80-bit and 112-bit security levels. To instantiate the CRH efficiently, we use an Ajtai-based SNARK-friendly collision-resistant hash function that is similar to the one used by Ben-Sasson et al. [21]. In our implementation, the modulus q is set to be the underlying SNARK implementation 254-bit field prime, and the dimension d is set to 3 for the 80-bit security level, and to 4 for the 112-bit security level based on the analysis in [71]. For PRFs and commitments, we use a hand-optimized implementation of SHA-256. Furthermore, we adopt the SNARK-friendly primitives for encryption used proposed in Appendix B [71], in which an efficient circuit for hybrid encryption in the case of 80-bit security level was proposed. The circuit performs the public key operations in a prime-order subgroup of the Galois field extension $\mathbb{F}_{p^\mu}$, where $\mu = 4$, p is the

underlying SNARK field prime (typically 254-bit prime, i.e. p^{μ} is over 1000-bit), and the prime order of the subgroup used is 398-bit prime. This was originally inspired by Pinocchio coin [6]. The circuit then applies a lightweight cipher like Speck [72] or Chaskey-LTS [73] with a 128-bit key to perform symmetric encryption in the CBC mode, as using the standard AES-128 instead will result in a higher cost [71] [2]. For the 112-bit security, using the same method for public key operations requires intensive factorization to find suitable parameters, therefore we use a manually optimized RSA-OAEP encryption circuit with a 2048-bit key instead (Note that this can be improved by our optimizations for RSA (Chapter 3) or using our SNARK-friendly Elliptic curve (Appendix B), which both appeared after this work).

In the next section, we will illustrate how using SNARK-friendly implementations can lead to **2.0-3.7**× savings in the size of the circuits at the 80-bit security level, compared to the case when naive straightforward implementation are used. We will also illustrate that the performance is also practical in the higher security level case.

Optimizations for `finalize`. In addition to the SNARK-friendly optimizations, we focus on optimizing the $O(N)$-sized `finalize` circuit since this is our main performance bottleneck. All other SNARK proofs in our scheme are for $O(1)$-sized circuits. Two key observations allow us to greatly improve the performance of the proof generation during `finalize`.

[2] This work was prior to developing our efficient AES circuit illustrated in Chapter 3

Optimization 1: Minimize Simulation Sound Extractable NIZKs. First, we observe that in the security proof [2], the simulator need not extract any new witnesses when a corrupted manager submits proofs during a `finalize` operation. All witnesses necessary will have been learned or extracted by the simulator at this point. Therefore, we can employ an ordinary SNARK instead of a stronger simulation sound extractable NIZK during `finalize`. For `freeze` and `compute`, we still use the stronger NIZK. This optimization reduces our SNARK circuit sizes by **1.5**× as can be inferred from Figure 4.10 of Section 4.5, after SNARK-friendly optimizations are applied.

Optimization 2: Minimize public-key encryption in SNARKs. Second, during `finalize`, the manager encrypts each party $\mathcal{P}_i$'s output coins to $\mathcal{P}_i$'s key, resulting in a ciphertext ct_i. The ciphertexts $\{\mathsf{ct}_i\}_{i\in[N]}$ would then be submitted to the contract along with appropriate SNARK proofs of correctness. Here, if a public-key encryption is employed to generate the ct_i's, it would result in relatively large SNARK circuit size. Instead, we rely on a symmetric-key encryption scheme denoted SENC in Figures 4.5 and 4.7. This requires that the manager and each $\mathcal{P}_i$ perform a key exchange to establish a symmetric key k_i. During an `compute`, the user encrypts this k_i to the manager's public key $\mathcal{P}_{\mathcal{M}}.\mathsf{epk}$, and prove that the k encrypted is consistent with the k committed to earlier in cm_i. The SNARK proof during `finalize` now only needs to include commitments and symmetric encryptions instead of public key encryptions in the circuit – the latter much more expensive.

This second optimization additionally gains us a factor of **1.9**× as shown in

Figure 4.10 of Section 4.5 after applying the previous optimizations. Overall, all optimizations will lead to a gain of more than **10**× in the `finalize` circuit.

Remarks about the common reference string. SNARK schemes require the generation of a common reference string (CRS) during a pre-processing step. This common reference string consists of an evaluation key for the prover, and a verification key for the verifier. Unless we employ recursively composed SNARKs [12, 69] whose costs are significantly higher, the evaluation key is circuit-dependent, and its size is proportional to the circuit's size. In comparison, the verification key is $O(|\mathsf{in}| + |\mathsf{out}|)$ in size, i.e., depends on the total length of inputs and outputs, but independent of the circuit size. Note that *only the verification key portion of the CRS needs to be included in the public contract that lives on the blockchain.*

We remark that the CRS for protocol $\mathsf{UserP}_{\mathrm{cash}}$ is shared globally, and can be generated in a one-time setup. In comparison, the CRS for each HAWK contract would depend on the HAWK contract, and therefore exists per instance of HAWK contract. To minimize the trust necessary in the CRS generation, one can employ either trusted hardware or use secure multi-party computation techniques as described by Ben-Sasson et al. [74].

Finally, in the future when new primitives become sufficiently fast, it is possible to drop-in and replace our SNARKs with other primitives that do not require per-circuit preprocessing.

4.5 Implementation and Evaluation

4.5.1 Compiler Implementation

Our compiler consists of several steps, which we illustrate in Figure 4.8 and describe below:

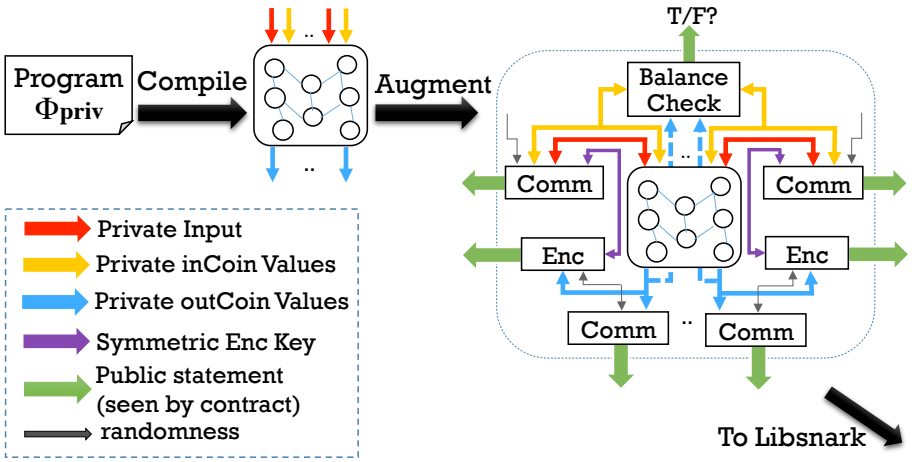

Figure 4.8: Compiler overview. Circuit augmentation for `finalize`.

Preprocessing: First, the input HAWK program is split into its *public contract* and *private contract* components. The public contract is Serpent code, and can be executed directly atop an ordinary cryptocurrency platform such as Ethereum. The private contract is written in a subset of the C language, and is passed as input to the Pinocchio arithmetic circuit compiler [4]. Currently, our private contract inherits the limitations of the Pinocchio compiler, e.g., cannot support dynamic-length loops. In the future, we can relax these limitations by employing recursively composition of SNARKs.

Circuit Augmentation: After compiling the preprocessed private contract code with Pinocchio, we have an arithmetic circuit representing the input/output relation ϕ_{priv}. This becomes a subcomponent of a larger arithmetic circuit, which we assemble using a customized circuit assembly tool. This tool is parameterized by the number of parties and the input/output datatypes, and attaches cryptographic constraints, such as computing commitments and encryptions over each party's output value, and asserting that the input and output values satisfy the balance property.

Cryptographic Protocol: Finally, the augmented arithmetic circuit is used as input to a state-of-the-art zkSNARK library, `libsnark` [5]. To avoid implementing SNARK verification in Ethereum's Serpent language, a SNARK verification opcode must be added to Ethereum's stack machine. We finally compile an executable program for the parties to compute the libsnark proofs according to our protocol.

4.5.2 Additional Examples

Besides our running example of a sealed-bid auction (Figure 4.1), we implemented several other examples in HAWK, demonstrating various capabilities:

Crowdfunding: A Kickstarter-style crowdfunding campaign, (also known as an assurance contract in economics literature [75]) overcomes the "free-rider problem," allowing a large number of parties to contribute funds towards some social good. If the minimum donation target is reached before the deadline,

then the donations are transferred to a designated party (the entrepreneur); otherwise, the donations are refunded. HAWK preserves privacy in the following sense: a) the donations pledged are kept private until the deadline; and b) if the contract fails, only the manager learns the amount by which the donations were insufficient. These privacy properties may conceivably have a positive effect on the willingness of entrepreneurs to launch a crowdfund campaign and its likelihood of success.

Rock Paper Scissors: A two-player lottery game, and naturally generalized to an N-player version. Our HAWK implementation provides the same notion of financial fairness as in [29, 30] and provides stronger security/privacy guarantees. If any party (including the manager), cheats or aborts, the remaining honest parties receive the maximum amount they might have won otherwise. Furthermore, we go beyond prior works [29, 30] by concealing the players' moves and the pseudonym of the winner to everyone except the manager.

"Swap" Financial Instrument: An individual with a risky investment portfolio (e.g, one who owns a large number of Bitcoins) may hedge his risks by purchasing insurance (e.g., by effectively betting against the price of Bitcoin with another individual). Our example implements a simple swap instrument where the price of a stock at some future date (as reported by a trusted authority specified in the public contract) determines which of two parties receives a payout. The private contract ensures the privacy of both the details of the agreement (i.e., the price threshold) and the outcome.

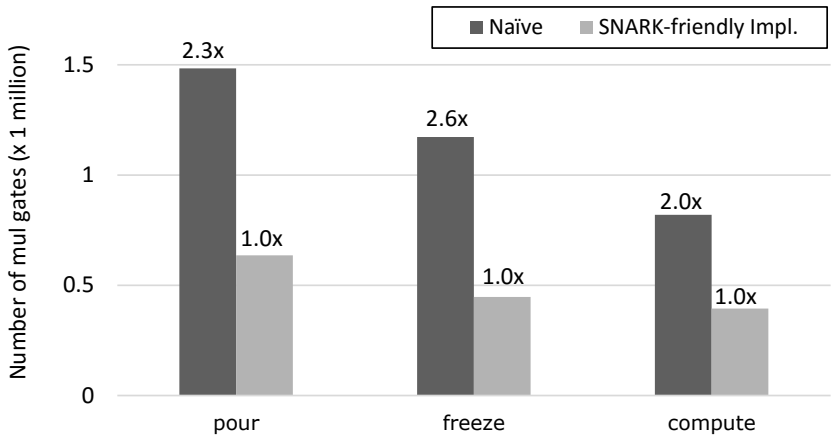

Figure 4.9: **Gains of using SNARK-friendly implementation for the user-side circuits:** `pour`, `freeze` and `compute` at 80-bit security.

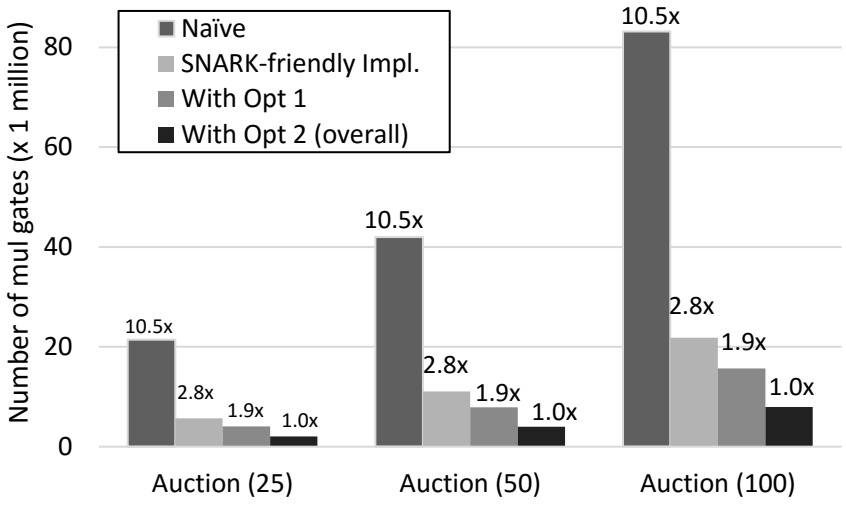

Figure 4.10: **Gains after adding each optimization to the `finalize` auction circuit, with 25, 50 and 100 Bidders.** Opt 1 and Opt 2 are two practical optimizations detailed in Section 4.4.

Table 4.1: **Performance of the zk-SNARK circuits for the user-side circuits:** `pour`, `freeze` and `compute` (same for all applications). `MUL` denotes multiple (4) cores, and `ONE` denotes a single core. The `mint` operation does not involve any SNARKs, and can be computed within tens of microseconds. The Proof includes any additional cryptographic material used for the SNARK-lifting transformation.

		80-bit security			112-bit security		
		`pour`	`freeze`	`compute`	`pour`	`freeze`	`compute`
KeyGen(s)	`MUL`	26.3	18.2	15.9	36.7	30.5	34.6
	`ONE`	88.2	63.3	54.42	137.2	111.1	131.8
Prove(s)	`MUL`	12.4	8.4	9.3	18.5	15.7	16.8
	`ONE`	27.5	20.7	22.5	42.2	40.5	41.7
Verify(ms)		9.7	9.1	10.0	9.9	9.3	9.9
EvalKey(MB)		148	106	90	236	189	224
VerKey(KB)		7.3	4.4	7.8	8.7	5.3	8.4
Proof(KB)		0.68	0.68	0.68	0.71	0.71	0.71
Stmt(KB)		0.48	0.16	0.53	0.57	0.19	0.53

4.5.3 Performance Evaluation

We evaluated the performance for various examples, using an Amazon EC2 `r3.8xlarge` virtual machine. We assume a maximum of 2^{64} leaves for the Merkle trees, and we present results for both 80-bit and 112-bit security levels. Our benchmarks actually consume at most 27GB of memory and 4 cores in the most expensive case. Tables 4.1 and 4.2 illustrate the results – we focus on evaluating the zk-SNARK performance since all other computation time is negligible in comparison. We highlight some important observations:

- **On-chain computation** (dominated by zk-SNARK verification time) is very small in all cases, ranging from **9 to 20 milliseconds** The running time of the verification algorithm is just linearly dependent on the size of the public

Table 4.2: **Performance of the zk-SNARK circuits for the manager circuit `finalize` for different applications. The manager circuits are the same for both security levels.** `MUL` denotes multiple (4) cores, and `ONE` denotes a single core.

	swap	rps	auction		crowdfund	
#Parties	2	2	10	100	10	100
KeyGen(s) MUL	8.6	8.0	32.3	300.4	32.16	298.1
ONE	27.8	24.9	124	996.3	124.4	976.5
Prove(s) MUL	3.2	3.1	15.4	169.3	15.2	169.2
ONE	7.6	7.4	40.1	384.2	40.3	377.5
Verify(ms)	8.4	8.4	10	19.9	10	19.8
EvalKey(GB)	0.04	0.04	0.21	1.92	0.21	1.91
VerKey(KB)	3.3	2.9	12.9	113.8	12.9	113.8
Proof(KB)	0.28	0.28	0.28	0.28	0.28	0.28
Stmt(KB)	0.23	0.2	1.11	10.26	1.11	10.26

statement, which is far smaller than the size of the computation, resulting into small verification time.

- **On-chain public parameters**: As mentioned in Section 4.3.3, not the entire SNARK common reference string (CRS) need to be on the blockchain, but only the verification key part of the CRS needs to be on-chain. Our implementation suggests the following: the private cash protocol requires a verification key of 23KB to be stored on-chain – this verification key is globally shared and there is only a single instance. Besides the globally shared public parameters, each HAWK contract will additionally require **13-114 KB** of verification key to be stored on-chain, for 10 to 100 users. This per-contract verification key is circuit-dependent, i.e., depends on the contract program. We refer the readers to Section 4.3.3 for more discussions on techniques for performing trusted

setup.

- **Manager computation:** Running private auction or crowdfunding protocols with 100 participants requires under 6.5min proof time for the manager on a single core, and under **2.85min** on 4 cores. This translates to under **$0.14** of EC2 time [76].

- **User computation:** Users' proof times for `pour`, `freeze` and `compute` are under one minute, and independent of the number of parties. Additionally, in the worst case, the peak memory usage of the user is less than 4 GB.

Savings from protocol optimizations. Figure 4.9 illustrates the performance gains attained by using a SNARK-friendly implementation for the user-side circuits, i.e. `pour, freeze` and `compute` w.r.t. the naive implementation at the 80-bit security level. We calculate the naive implementation cost using conservative estimates for the straightforward implementation of standard cryptographic primitives. The figure shows a gain of **2.0-2.6**× compared to the naive implementation. Furthermore, Figure 4.10 illustrates the performance gains attained by our protocol optimizations described in Section 4.4 The figure considers the sealed-bid auction finalize circuit at different number of bidders. We show that the SNARK-friendly implementation along with our two optimizations combined significantly reduce the SNARK circuit sizes, and achieve a gain of **10**× relative to a straightforward implementation. The figure also illustrates that the manager's cost is proportional to the number of participants. (By contrast, the user-side costs are independent of the number of participants).

4.6 Conclusion and Future Directions

In this chapter, we presented HAWK a system for privacy-preserving smart contracts. HAWK relies on zk-SNARKs as a main tool for verifying computations outside the blockchain in a privacy-preserving manner, and was built using our tools for low-level circuit construction and composable zero-knowledge proofs presented in Appendices A and B. The performance results presented in this chapter can be further improved on the circuit level by the optimizations in Chapter 3, and on the zero knowledge cryptographic back end level through the recent work by Groth et al [77,78]. Furthermore, one limitation for our HAWK system is the trusted preprocessing phase needed per contract. In Chapter 6, we discuss further directions to look into based on new zero knowledge proof systems.

Chapter 5: Investigating the future of Criminal Smart Contracts[1]

Cryptocurrencies such as Bitcoin remove the need for trusted third parties from basic monetary transactions and offer anonymous (more accurately, pseudonymous) transactions between individuals. While attractive for many applications, these features have a dark side. Bitcoin has stimulated the growth of ransomware [79], money laundering [80], and illicit commerce, as exemplified by the notorious Silk Road [81].

New cryptocurrencies such as Ethereum (as well as systems such as Counterparty [82] and SmartContract [83]) offer even richer functionality than Bitcoin. In a fully distributed system such as Ethereum, smart contracts enable general fair exchange (atomic swaps) without a trusted third party, and thus can effectively guarantee payment for successfully delivered data or services. Given the flexibility of such smart contract systems, it is to be expected that they will stimulate not just new beneficial services, but new forms of crime.

We refer to smart contracts that facilitate crimes in distributed smart contract systems as *criminal smart contracts* (CSCs). An example of a CSC is a smart contract for (private-)key theft. Such a CSC might pay a reward for (confidential)

[1]Based on a joint work with Ari Juels and Elaine Shi, that appeared in the ACM Conference on Computer and Communications Security (CCS), 2016 [8].

delivery of a target key sk, such as a certificate authority's private digital signature key.

We explore the following key questions in this chapter. *Could CSCs and new verifible computation techniques enable a wider range of new crimes than earlier cryptocurrencies (Bitcoin)? How practical will such new crimes be?* And *What key advantages do CSCs provide to criminals compared with conventional online systems?* Exploring these questions is essential to identifying threats and devising countermeasures.

Would-be criminals face two basic challenges in the construction of CSCs. First, it is not immediately obvious whether a CSC is at all feasible for a given crime, such as key theft. This is because it is challenging to ensure that a CSC achieves a key property in this work that we call *commission-fair*, meaning informally that its execution guarantees *both* commission of a crime and commensurate payment for the perpetrator of the crime or *neither*. Fair exchange is necessary to ensure commission-fairness, but not sufficient: We show how CSC constructions implementing fair exchange still allow a party to a CSC to cheat. Correct construction of CSCs can thus be delicate.

Second, even if a CSC can in principle be constructed, given the limited op-codes in existing smart contract systems (such as Ethereum), it is not immediately clear that the CSC can be made *practical*. By this we mean that the CSC can be executed without unduly burdensome computational effort, which in some smart contract systems (e.g., Ethereum) would also mean unacceptably high execution fees levied against the CSC.

The following example illustrates these challenges.

Example 1a (Key compromise contract)**.** Contractor $\mathcal{C}$ posts a request for theft and delivery of the signing key $\mathsf{sk}_{\mathcal{V}}$ of a victim certificate authority (CA) CertoMart. $\mathcal{C}$ offers a reward $\$\mathsf{reward}$ to a perpetrator $\mathcal{P}$ for (confidentially) delivering the CertoMart private key $\mathsf{sk}_{\mathcal{V}}$ to $\mathcal{C}$.

To ensure fair exchange of the key and reward in Bitcoin, $\mathcal{C}$ and $\mathcal{P}$ would need to use a trusted third party or communicate directly, raising the risks of being cheated or discovered by law enforcement. They could vet one another using a reputation system, but such systems are often infiltrated by law enforcement authorities [38]. In contrast, a decentralized smart contract can achieve self-enforcing fair exchange. For key theft, this is possible using the CSC Key-Theft in the following example:

Example 1b (Key compromise CSC)**.** *$\mathcal{C}$ generates a private / public key pair $(\mathsf{sk}_{\mathcal{C}}, \mathsf{pk}_{\mathcal{C}})$ and initializes* Key-Theft *with public keys $\mathsf{pk}_{\mathcal{C}}$ and $\mathsf{pk}_{\mathcal{V}}$ (the CertoMart public key).* Key-Theft *awaits input from a claimed perpetrator $\mathcal{P}$ of a pair (ct, π), where π is a zero-knowledge proof that $\mathsf{ct} = \mathit{enc}_{\mathsf{pk}_{\mathcal{C}}}[sk_{\mathcal{V}}]$ is well-formed.* Key-Theft *then verifies π and upon success sends a reward of $\$\mathsf{reward}$ to $\mathcal{P}$. The contractor $\mathcal{C}$ can then download and decrypt ct to obtain the compromised key $\mathsf{sk}_{\mathcal{V}}$.*

Key-Theft implements a fair exchange between $\mathcal{C}$ and $\mathcal{P}$, paying a reward to $\mathcal{P}$ *if and only if* $\mathcal{P}$ delivers a valid key (as proven by π), eliminating the need for a trusted third party. But it is *not commission-fair*, as it does not ensure that sk_{vict}

actually has value. The CertoMart can neutralize the contract by preemptively revoking its own certificate and then itself claiming $\mathcal{C}$'s reward $\$\mathsf{reward}!$

As noted, a major thrust of this work is showing how, for CSCs such as Key-Theft, criminals will be able to bypass such problems and still construct commission-fair CSCs. (For key compromise, it is necessary to enable contract cancellation should a key be revoked.) Additionally, we show that these CSCs can be efficiently realized using existing cryptocurrency tools or features currently envisioned for cryptocurrencies (e.g., zk-SNARKS [4, 5]).

In this chapter, we show that it is or will be possible in smart contract systems to construct CSCs for two types of crime:

1. Leakage / sale of secret documents;

2. Theft of private keys.

In the full version of the paper [8], we also study the applicability of "*Calling-card*" crimes, a broad class of physical-world crimes (murder, arson, etc.) using smart contracts, however, we do not cover them as they do not rely on verifiable computation. Additionally, we study CSCs using trusted hardware for password theft.

The fact that CSCs are possible in principle is not surprising. Previously, however, it was not clear how practical or extensively applicable CSCs might be. As our constructions for commission-fair CSCs show, constructing CSCs is not as straightforward as it might seem, but new cryptographic techniques and new approaches to smart contract design can render them feasible and even practical.

Our work therefore shows how imperative it is for the community to consider the construction of defenses against CSCs. Criminal activity committed under the guise of anonymity has posed a major impediment to adoption for Bitcoin. Yet there has been little discussion of criminal contracts in public forums on cryptocurrency [84] and the launch of Ethereum took place in July 2015. *It is only by recognizing CSCs early in their lifecycle that the community can develop timely countermeasures to them,* and see the promise of distributed smart contract systems fully realized.

While our focus is on preventing evil, happily the techniques we propose can also be used to create beneficial contracts. We explore both techniques for structuring CSCs and the use of cutting-edge cryptographic tools, e.g., Succinct Non-interactive ARguments of Knowledge (SNARKs), in CSCs. Like the design of beneficial smart contracts, CSC construction requires a careful combination of cryptography with commission-fair design [85].

In summary, our contributions are:

- *Criminal smart contracts:* We initiate the study of CSCs as enabled by Turing-complete scripting languages in next-generation cryptocurrencies. We explore CSCs for two different types of crimes: leakage of secrets in Section 5.2 (e.g., pre-release Hollywood films) and key compromise / theft (of, e.g., a CA signing key) in Section 5.3. We explore the challenges involved in crafting such criminal contracts and demonstrate (anticipate) new techniques to resist neutralization and achieve commission-fairness.

- *Proof of concept:* To demonstrate that even sophisticated CSC are realistic, we report (in their respective sections) on implementation of the CSCs we explore. Our CSC for leakage of secrets is *efficiently realizable today* in existing smart contract languages (e.g., that of Ethereum). The key theft CSC relies respectively for efficiency and realizability on features currently envisioned by the cryptocurrency community.

  They too, however, are within practical reach as shown for example for our key-theft CSC, which relies on zk-SNARKs. Our experiments show that verification—the most important function, as it is performed by all full nodes—requires as little as 9.9 msec on a 288-byte proof (with execution on an Amazon EC2 r3.2xlarge instance with 2.5 GHz processors).

- *Countermeasures:* We briefly discuss in Section 5.4 how our work can help prevent a proliferation of CSCs. Briefly, to be most effective, CSCs must be advertised, making them detectible given community vigilance. Miners have an economic incentive not to include CSC transactions in blocks, as CSCs degrade the market value of a cryptocurrency. Consequently, awareness and robust detection strategies may offer an effective general defense. A key contribution of this work is to show the need for such countermeasures and stimulate exploration of their implementation in smart contract systems such as Ethereum.

5.1 Notation and Threat Model

In this section, we describe the notations and the threat model we assume. It is mostly similar to Section 4.2, but with additional notes related to our protocols in the chapter. As in Chapter 4, we consider the setting of a public blockchain system with support for smart contracts, like Ethereum [19].

Protocols in the smart contract model. Our model treats a *contract* as a special party that is *entrusted to enforce correctness but not privacy*, as noted above. (In reality, of course, a contract is enforced by the network.) All messages sent to the contract and its internal state are publicly visible. A contract interacts with users and other contracts by exchanging messages (also referred to as transactions). Money, expressed in the form of account balances, is recorded in the global ledger. Contracts can access and update the ledger to implement money transfers between users, who are represented by pseudonymous public keys.

5.1.1 Threat Model

We adopt the following threat model in this work.

- *Blockchain: Trusted for correctness but not privacy.* We assume that the blockchain always correctly stores data and performs computations and is always available. The blockchain exposes all of its internal states to the public, however, and retains no private data.

- *Arbitrarily malicious contractual parties.* We assume that contractual parties are mutually distrustful, and they act solely to maximize their own benefit. Not only can they deviate arbitrarily from the prescribed protocol, they can also abort from the protocol prematurely.

- *Network influence of the adversary.* We assume that messages between the blockchain and players are delivered within a bounded delay, i.e., not permanently dropped. (A player can always resend a transaction dropped by a malicious miner.) In our model, an adversary immediately receives and can arbitrarily reorder messages, however. In real-life decentralized cryptocurrencies, the winning miner sets the order of message processing. An adversary may collude with certain miners or influence message-propagation among nodes. As we show in Section 5.3, for key-theft contracts, message-reordering enables a rushing attack that a commission-fair CSC must prevent.

5.1.2 Notational Conventions

We now explain some notational conventions for writing contracts.

- **Currency and ledger.** We use $\mathsf{ledger}[\mathcal{P}]$ to denote party $\mathcal{P}$'s balance in the global ledger. For clarity, variables that begin with a \$ sign denote money, but otherwise behave like ordinary variables.

  Unlike in Ethereum's Serpent language, in our formal notation, when a contract receives some $\$\mathsf{amount}$ from a party $\mathcal{P}$, this is only message transfer, and no currency transfer has taken place at this point. Money transfers *only take*

effect when the contract performs operations on the ledger, denoted ledger.

- **Pseudonymity.** Parties can use pseudonyms to obtain better anonymity. In particular, a party can generate arbitrarily many public keys. In our notational system, when we refer to a party $\mathcal{P}$, $\mathcal{P}$ denotes the party's pseudonym. The formal blockchain model [2] we adopt provides a contract wrapper that manages the pseudonym generation and the message signing necessary for establishing an authenticated channel to the contract. These details are abstracted away from the main contract program.

- **Timer.** Time progresses in rounds. At the beginning of each round, the contract's **Timer** function will be invoked. The variable T encodes the current time.

- **Entry points and variable scope.** A contract can have various entry points, each of which is invoked when receiving a corresponding message type. Thus entry points behave like function calls invoked upon receipt of messages.

  All variables are assumed to be globally scoped, with the following exception: When an entry point says "Upon receiving a message from *some* party $\mathcal{P}$," this allows the registration of a new party $\mathcal{P}$. In general, contracts are open to any party who interacts with them. When a message is received from $\mathcal{P}$ (without the keyword "some"), party $\mathcal{P}$ denotes a fixed party – and a well-formed contract has already defined $\mathcal{P}$.

5.2 CSCs for Leakage of Secrets

As a first example of the power of smart contracts, we show how an existing type of criminal contract deployed over Bitcoin can be made more robust and functionally enhanced as a smart contract and can be practically implemented in Ethereum.

Among the illicit practices stimulated by Bitcoin is payment-incentivized *leakage*, i.e., public disclosure, of secrets. The recently created web site Darkleaks [86] (a kind of subsidized Wikileaks) serves as a decentralized market for crowdfunded public leakage of a wide variety of secrets, including, "Hollywood movies, trade secrets, government secrets, proprietary source code, industrial designs like medicine or defence, [etc.]."

Intuitively, we define commission-fairness in this setting to mean that a contractor $\mathcal{C}$ receives payment iff it leaks a secret in its entirety within a specified time limit.

As we show, Darkleaks highlights the inability of Bitcoin to support commission-fairness. We show how a CSC can in fact achieve commission-fairness with high probability.

5.2.1 Darkleaks

In the Darkleaks system, a contractor $\mathcal{C}$ who wishes to sell a piece of content M partitions it into a sequence of n segments $\{m_i\}_{i=1}^{n}$. At a time (block height) T_{open} pre-specified by $\mathcal{C}$, a randomly selected subset $\Omega \subset [n]$ of k segments is publicly

disclosed as a sample to entice donors / purchasers—those who will contribute to the purchase of M for public leakage. When $\mathcal{C}$ determines that donors have collectively paid a sufficient price, $\mathcal{C}$ decrypts the remaining segments for public release. The parameter triple (n, k, T_{open}) is set by $\mathcal{C}$ (where $n = 100$ and $k = 20$ are recommended defaults).

To ensure a fair exchange of M for payment without direct interaction between parties, Darkleaks implements a (clever) protocol on top of the Bitcoin scripting language. The main idea is that for a given segment m_i of M that is not revealed as a sample in Ω, donors make payment to a Bitcoin account a_i with public key pk_i. The segment m_i is encrypted under a key $\kappa = H(\mathsf{pk}_i)$ (where H = SHA-256). To spend its reward from account a_i, $\mathcal{C}$ is forced by the Bitcoin transaction protocol to disclose pk_i; thus the act of *spending the reward automatically enables the community to decrypt* m_i.

We present an overview of the existing, broken Darkleaks protocol, as we are unaware of any unified technical presentation elsewhere. (Specific details, e.g., message formats, may be found in the Darkleaks source code [86], and cryptographic primitives h_1, h_2, h_2, and $(\mathsf{enc}, \mathsf{dec})$ are specified below.)

The protocol steps are as follows:

- *Create:* The contractor $\mathcal{C}$ partitions the secret $M = m_1 \parallel m_2 \parallel \ldots \parallel m_n$. For each segment m_i in $M = \{m_i\}_{i=1}^{n}$, $\mathcal{C}$ computes:
  - A Bitcoin (ECDSA) private key $\mathsf{sk}_i = h_1(m_i)$ and the corresponding public key pk_i.

– The Bitcoin address $a_i = h_2(\mathsf{pk}_i)$ associated with pk_i.

– A symmetric key $\kappa_i = h_3(\mathsf{pk}_i)$, computed as a hash of public key pk_i.

– The ciphertext $e_i = \mathsf{enc}_{\kappa_i}[m_i]$.

$\mathcal{C}$ publishes: The parameter triple (n, k, T_{open}), ciphertexts $E = \{e_i\}_{i=1}^{n}$, and Bitcoin addresses $A = \{a_i\}_{i=1}^{n}$.

- *Challenge:* At epoch (block height) T_{open}, the current Bitcoin block hash B_t serves as a pseudorandom seed for a challenge $S^* = \{s_i\}_{i=1}^{k}$.

- *Response:* In epoch T_{open}, $\mathcal{C}$ publishes the subset of public keys $PK^* = \{pk_s\}_{s \in S^*}$ corresponding to addresses $A^* = \{a_s\}_{s \in S^*}$. (The sample of segments $M^* = \{m_s\}_{s \in S^*}$ can then be decrypted by the Darkleaks community.)

- *Payment:* To pay for M, buyers send Bitcoin to the addresses $A - A^*$ corresponding to unopened segments.

- *Disclosure:* The leaker $\mathcal{C}$ claims the payments made to addresses in $A - A^*$. As spending the Bitcoin in address a_i discloses pk_i., *decryption of all unopened segments $M - M^*$ is automatically made possible for the Darkleaks community.*

Here, $h_1 =$ SHA-256, $h_2 =$ RIPEMD-160(SHA-256()), and $h_3 =$ SHA-256(SHA-256()). The pair $(\mathsf{enc}, \mathsf{dec})$ in Darkleaks corresponds to AES-256-ECB.

As a byproduct of its release of PK^* in response to challenge S^*, $\mathcal{C}$ proves (weakly) that undecrypted ciphertexts are well-formed, i.e., that $e_i = \mathsf{enc}_{\kappa_i}[m_i]$ for

$\kappa_i = h_3(pk_i)$. This cut-and-choose-type proof assures buyers that when $\mathcal{C}$ claims its reward, M will be fully disclosed.

Shortcomings and vulnerabilities. The Darkleaks protocol has three major shortcomings / vulnerabilities that appear to stem from fundamental functional limitations of Bitcoin's scripting language when constructing contracts without direct communication between parties. The first two undermine commission-fairness, while the third limits functionality.

1. *Delayed release:* $\mathcal{C}$ can refrain from spending purchasers' / donors' payments and releasing unopened segments of M until after M loses value. E.g., $\mathcal{C}$ could withhold segments of a film until after its release in theaters, of an industrial design until after it is produced, etc.

2. *Selective withholding:* $\mathcal{C}$ can choose to forego payment for selected segments and not disclose them. For example, $\mathcal{C}$ could leak and collect payment for all of a leaked film but the last few minutes (which, with high probability, will not appear in the sample Ω), significantly diminishing the value of leaked segments.

3. *Public leakage only:* Darkleaks can only serve to leak secrets *publicly*. It does not enable fair exchange for *private leakage*, i.e., for payment in exchange for a secret M encrypted under the public key of a purchaser $\mathcal{P}$.

Additionally, Darkleaks has a basic protocol flaw:

4. *Reward theft:* In the Darkleaks protocol, the Bitcoin private key sk_i corresponding to pk_i is derived from m_i; specifically $\mathsf{sk}_i = \text{SHA-256}(m_i)$. Thus, the source of M

(e.g., the victimized owner of a leaked film) can derive sk_i and steal rewards received by $\mathcal{C}$. (Also, when $\mathcal{C}$ claims a reward, a malicious node that receives the transaction can decrypt m_i, compute $\mathsf{sk}_i = \text{SHA-256}(m_i)$, and potentially steal the reward by flooding the network with a competing transaction [87].)

This last problem is easily remedied by generating the set $\{\kappa_i\}_{i=1}^n$ of segment encryption keys pseudorandomly or randomly, which we do in our CSC designs.

Remark: In *any* protocol in which goods are represented by a random sample, not just Darkleaks, $\mathcal{C}$ can insert a small number of valueless or duplicate segments into M. With non-negligible probability, these will not result in an invalid-looking sample Ω, so Ω necessarily provides only a *weak* guarantee of the global validity of M. The larger k and n, the smaller the risk of such attack. Formal analysis of human-verified proofs of this kind and/or ways of automating them is an interesting problem beyond the scope of this work, but important in assessing end-to-end security in a CSC of this kind.

5.2.2 A generic public-leakage CSC

We now present a smart contract that realizes public leakage of secrets using blackbox cryptographic primitives. (We later present efficient realizations.) This contract overcomes limitation 1. of the Darkleaks protocol (delayed release) by enforcing disclosure of M at a pre-specified time T_{end}—or else immediately refunding buyers' money. It addresses limitation 2. (selective withholding) by ensuring that M is revealed in an all-or-nothing manner. (We later explain how to achieve private

leakage and overcome limitation 3.)

Again, we consider settings where $\mathcal{C}$ aims to sell M for public release after revealing sample segments M^*.

Informal protocol description. Informally, the protocol involves the following steps:

- *Create contract.* A seller $\mathcal{C}$ initializes a smart contract with the encryption of a randomly generated *master secret key* msk. The master secret key is used to generate (symmetric) encryption keys for the segments $\{m_i\}_{i=1}^n$. $\mathcal{C}$ provides a cryptographic commitment $c_0 := \mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0)$ of msk to the contract. (To meet the narrow technical requirements of our security proofs, the commitment is an encryption with randomness r_0 under a public key pk created during a trusted setup step.) The master secret key msk can be used to decrypt all leaked segments of M.

- *Upload encrypted data.* For each $i \in [n]$, $\mathcal{C}$ generates encryption key $\kappa_i := \mathsf{PRF}(\mathsf{msk}, i)$, and encrypts the i-th segment as $\mathsf{ct}_i = \mathsf{enc}_{\kappa_i}[m_i]$. $\mathcal{C}$ sends all encrypted segments $\{\mathsf{ct}_i\}_{i\in[n]}$ to the contract (or, for efficiency, provides hashes of copies stored with a storage provider, e.g., a peer-to-peer network). Interested purchasers / donors can download the segments of M, but cannot decrypt them yet.

- *Challenge.* The contract generates a random challenge set $\Omega \subset [n]$, in practice today in Ethereum could be based on the hash of a recent block. Another future possibility is some well known randomness source, e.g., the NIST ran-

domness beacon [88], perhaps relayed through an authenticated data feed.

- *Response.* $\mathcal{C}$ reveals the set $\{\kappa_i\}_{i\in\Omega}$ to the contract, and gives ZK proofs that the revealed secret keys $\{\kappa_i\}_{i\in\Omega}$ are generated correctly from the msk encrypted as c_0.

- *Collect donations.* During a donation period, potential purchasers / donors can use the revealed secret keys $\{\kappa_i\}_{i\in\Omega}$ to decrypt the corresponding segments. If they like the decrypted segments, they can donate money to the contract as contribution for the leakage.

- *Accept.* If enough money has been collected, $\mathcal{C}$ decommits msk for the contract (sends the randomness for the ciphertext along with msk). If the contract verifies the decommitment successfully, all donated money is paid to $\mathcal{C}$. The contract thus enforces a fair exchange of msk for money. (If the contract expires at time T_{end} without release of msk, all donations are refunded.)

The contract. Our proposed CSC PublicLeaks for implementing this public leakage protocol is given in Figure 5.1. The ideal functionaly and the proof are in the online version of the paper [8].

5.2.3 Optimizations and Ethereum implementation

The formally specified contract PublicLeaks uses generic cryptographic primitives in a black-box manner. We now give a practical, optimized version, relying on the random oracle model (ROM), that eliminates trusted setup, and also achieves

Contract $\mathsf{PublicLeaks}$

Init: Set $\mathsf{state} := \textsc{init}$, and $\mathsf{donations} := \{\}$. Let $\mathsf{crs} := \mathsf{KeyGen}_{\mathrm{nizk}}(1^\lambda)$, $\mathsf{pk} := \mathsf{KeyGen}_{\mathrm{enc}}(1^\lambda)$ denote hardcoded public parameters generated through a trusted setup.

Create: Upon receiving ("create", c_0, $\{\mathsf{ct}_i\}_{i=1}^n$, T_{end}) from some leaker $\mathcal{C}$:
Set $\mathsf{state} := \textsc{created}$.
Select a random subset $\Omega \subset [n]$ of size k, and send ("challenge", Ω) to $\mathcal{C}$.

Confirm: Upon receiving ("confirm", $\{(\kappa_i, \pi_i)\}_{i\in\Omega}$) from $\mathcal{C}$:
Assert $\mathsf{state} = \textsc{created}$.
Assert that $\forall i \in S$: π_i is a valid NIZK proof (under crs) for the following statement:

$$\exists(\mathsf{msk}, r_0),\ \text{s.t.}\ \ (c_0 = \mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0)) \\ \wedge\ (\kappa_i = \mathsf{PRF}(\mathsf{msk}, i))$$

Set $\mathsf{state} := \textsc{confirmed}$.

Donate: Upon receiving ("donate", $\mathsf{amt}) from some purchaser $\mathcal{P}$:
Assert $\mathsf{state} = \textsc{confirmed}$.
Assert $\mathsf{ledger}[\mathcal{P}] \geq \mathsf{amt}$.
Set $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] - \mathsf{amt}$.
$\mathsf{donations} := \mathsf{donations} \cup \{(\mathsf{amt}, \mathcal{P})\}$.

Accept: Upon receiving ("accept", msk, r_0) from $\mathcal{C}$:
Assert $\mathsf{state} = \textsc{confirmed}$
Assert $c_0 = \mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0)$
$\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \mathsf{sum}(\mathsf{donations})$
Send ("leak", msk) to all parties.
Set $\mathsf{state} := \textsc{aborted}$.

Timer: If $\mathsf{state} = \textsc{confirmed}$ and $T > T_{\mathrm{end}}$: $\forall(\mathsf{amt}, \mathcal{P}) \in \mathsf{donations}$: let $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \mathsf{amt}$. Set $\mathsf{state} := \textsc{aborted}$.

Figure 5.1: A contract $\mathsf{PublicLeaks}$ that leaks a secret M to the public in exchange for donations.

better efficiency and easy integration with Ethereum [19].

A practical optimization. During contract creation, $\mathcal{C}$ chooses random $\kappa_i \xleftarrow{\$} \{0,1\}^\lambda$ for $i \in [n]$, and computes

$$c_0 := \{H(\kappa_1, 1), \ldots, H(\kappa_n, n)\}.$$

The master secret key is simply $\mathsf{msk} := \{\kappa_1, \ldots, \kappa_n\}$, i.e., the set of hash pre-

images. As in PublicLeaks, each segment m_i will still be encrypted as $\mathsf{ct}_i := \mathsf{enc}_\kappa[m_i]$. (For technical reasons—to achieve simulatability in the security proof [8] $\mathsf{enc}_\kappa[m_i] = m_i \oplus [H(\kappa_i, 1, \text{“enc”}) \,||\, H(\kappa_i, 2, \text{“enc”}) \ldots,$

$|| \, H(\kappa_i, z, \text{“enc”})]$ for suitably large z.)

$\mathcal{C}$ submits c_0 to the smart contract. When challenged with the set Ω, $\mathcal{C}$ reveals $\{\kappa_i\}_{i\in\Omega}$ to the contract, which then verifies its correctness by hashing and comparing with c_0. To accept donations, $\mathcal{C}$ reveals the entire msk.

This optimized scheme is asymptotically less efficient than our generic, black-box construction PublicLeaks—as the master secret key scales linearly in the number of segments n. But for typical, realistic document set sizes in practice (e.g., $n = 100$, as recommended for Darkleaks), it is more efficient.

Ethereum-based implementation. To demonstrate the feasibility of implementing leakage contracts using currently available technology, we implemented a version of the contract PublicLeaks atop Ethereum [19], using the Serpent contract language [89]. We specify the full implementation in detail in Appendix C.2.

The version we implemented relies on the practical optimizations described above. As a technical matter, Ethereum does not appear at present to support timer-activated functions, so we implemented **Timer** in such a way that purchasers / donors make explicit withdrawals, rather than receiving automatic refunds.

This public leakage Ethereum contract is highly efficient, as it does not require expensive cryptographic operations. It mainly relies on hashing (SHA3-256) for random number generation and for verifying hash commitments. The total number

of storage entries (needed for encryption keys) and hashing operations is $O(n)$, where, again, Darkleaks recommends $n = 100$. (A hash function call in practice takes a few micro-seconds, e.g., 3.92 μsecs measured on a core i7 processor.)

5.2.4 Extension: private leakage

As noted above, shortcoming 3. of Darkleaks is its inability to support *private* leakage, in which $\mathcal{C}$ sells a secret exclusively to a purchaser $\mathcal{P}$.

However, using verifiable computation techniques as illustrated earlier in the thesis, PublicLeaks can be modified for this purpose. The basic idea is for $\mathcal{C}$ not to reveal msk directly, but to provide a ciphertext $\mathsf{ct} = \mathsf{enc}_{\mathsf{pk}_\mathcal{P}}[\mathsf{msk}]$ on msk to the contract for a purchaser $\mathcal{P}$, along with a proof that ct is correctly formed, with respect to earlier commitments.

5.3 A Key-Compromise CSC

Example 1b earlier described a CSC that rewards a perpetrator $\mathcal{P}$ for delivering to $\mathcal{C}$ the stolen key $sk_\mathcal{V}$ of a victim $\mathcal{V}$—in this case a certificate authority (CA) with public key $pk_\mathcal{V}$. Recall that $\mathcal{C}$ generates a private / public key encryption pair $(\mathsf{sk}_\mathcal{C}, \mathsf{pk}_\mathcal{C})$. The contract accepts as a claim by $\mathcal{P}$ a pair (ct, π). It sends reward \$reward to $\mathcal{P}$ if π is a valid proof that $\mathsf{ct} = \mathsf{enc}_{\mathsf{pk}_\mathcal{C}}[\mathsf{sk}_\mathcal{V}]$ and $\mathsf{sk}_\mathcal{V}$ is the private key corresponding to $\mathsf{pk}_\mathcal{V}$.

Intuitively, a key-theft contract is commission-fair if it rewards a perpetrator $\mathcal{P}$ for delivery of a private key that: (1) $\mathcal{P}$ was responsible for stealing and (2) Is

valid for a substantial period of time.

This form of contract can be used to solicit theft of any type of private key, e.g., the signing key of a CA, the private key for a SSL/TLS certificate, a PGP private key, etc. (Similar contracts could solicit abuse, but not full compromise of a private key, e.g., forged certificates.)

Figure 5.2 shows the contract of Example 1b in our notation for smart contracts. We let crs here denote a common reference string for a NIZK scheme and $\mathsf{match}(\mathsf{pk}_{\mathcal{V}}, \mathsf{sk}_{\mathcal{V}})$ denote an algorithm that verifies whether $\mathsf{sk}_{\mathcal{V}}$ is the corresponding private key for some public key $\mathsf{pk}_{\mathcal{V}}$ in a target public-key cryptosystem.

As noted above, this CSC is *not* commission-fair. Thus we refer to it as KeyTheft-Naive.We use KeyTheft-Naive as a helpful starting point for motivating and understanding the construction of a commission-fair contract proposed later, called KeyTheft.

Contract KeyTheft-Naive

Init: Set $\mathsf{state} := \textsc{init}$. Let $\mathsf{crs} := \mathsf{KeyGen}_{\mathrm{nizk}}(1^\lambda)$ denote a hard-coded NIZK common reference string generated during a trusted setup process.

Create: Upon receiving ("create", \$reward, $\mathsf{pk}_\mathcal{V}$, T_{end}) from some contractor $\mathcal{C} := (\mathsf{pk}_\mathcal{C}, \ldots)$:

Assert $\mathsf{state} = \textsc{init}$.
Assert $\mathsf{ledger}[\mathcal{C}] \geq$ \$reward.
$\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] -$ \$reward.
Set $\mathsf{state} := \textsc{created}$.

Claim: Upon receiving ("claim", ct, π) from some purported perpetrator $\mathcal{P}$:

Assert $\mathsf{state} = \textsc{created}$.
Assert that π is a valid NIZK proof (under crs) for the following statement:

$$\exists r, \mathsf{sk}_\mathcal{V} \text{ s.t. } \mathsf{ct} = \mathsf{Enc}(\mathsf{pk}_\mathcal{C}, (\mathsf{sk}_\mathcal{V}, \mathcal{P}), r) \text{ and } \mathsf{match}(\mathsf{pk}_\mathcal{V}, \mathsf{sk}_\mathcal{V}) = \texttt{true}$$

$\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] +$ \$reward.
Set $\mathsf{state} := \textsc{claimed}$.

Timer: If $\mathsf{state} = \textsc{created}$ and current time $T > T_{\mathrm{end}}$:

$\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] +$ \$reward
$\mathsf{state} := \textsc{aborted}$

Figure 5.2: A naïve, flawed key theft contract (lacking commission-fairness)

5.3.1 Flaws in KeyTheft-Naive

The contract KeyTheft-Naive fails to achieve commission-fairness due to two shortcomings.

Revoke-and-claim attack. The CA $\mathcal{V}$ can revoke the key $\mathsf{sk}_\mathcal{V}$ and then itself submit the key for payment. The CA then not only negates the value of the contract but actually profits from it! This *revoke-and-claim* attack demonstrates that KeyTheft-Naive is not commission-fair in the sense of ensuring the delivery of a *usable* private key sk_V.

Rushing attack. Another attack is a rushing attack. As noted in Section 5.1, an adversary can arbitrarily reorder messages—a reflection of possible attacks against

the network layer in a cryptocurrency. (See also the formal blockchain model [2].) Thus, given a valid claim from perpetrator $\mathcal{P}$, a corrupt $\mathcal{C}$ can decrypt and learn $\mathsf{sk}_{\mathcal{V}}$, construct another valid-looking claim of its own, and make its own claim arrive before the valid one.

5.3.2 Fixing flaws in KeyTheft-Naive

We now show how to modify KeyTheft-Naive to prevent the above two attacks and achieve commission-fairness.

Thwarting revoke-and-claim attacks. In a revoke-and-claim attack against KeyTheft-Naive, $\mathcal{V}$ preemptively revokes its public key $\mathsf{pk}_{\mathcal{V}}$ and replaces it with a fresh one $\mathsf{pk}'_{\mathcal{V}}$. As noted above, the victim can then play the role of perpetrator $\mathcal{P}$, submit $\mathsf{sk}_{\mathcal{V}}$ to the contract and claim the reward. The result is that $\mathcal{C}$ pays \$reward to $\mathcal{V}$ and obtains a stale key.

We address this problem by adding to the contract a feature called *reward truncation*, whereby the contract accepts evidence of revocation Π_{revoke}.

This evidence Π_{revoke} can be an Online Certificate Status Protocol (OCSP) response indicating that $\mathsf{pk}_{\mathcal{V}}$ is no longer valid, a new certificate for $\mathcal{V}$ that was unknown at the time of contract creation (and thus not stored in Contract), or a certificate revocation list (CRL) containing the certificate with $\mathsf{pk}_{\mathcal{V}}$.

$\mathcal{C}$ could submit Π_{revoke}, but to minimize interaction by $\mathcal{C}$, KeyTheft could provide a reward \$smallreward to a third-party submitter. The reward could be small, as Π_{revoke} would be easy for ordinary users to obtain.

The contract then provides a reward based on the interval of time over which the key $\mathsf{sk}_{\mathcal{V}}$ remains valid. Let T_{claim} denote the time at which the key $\mathsf{sk}_{\mathcal{V}}$ is provided and T_{end} be an expiration time for the contract (which must not exceed the expiration of the certificate containing the targeted key). Let T_{revoke} be the time at which Π_{revoke} is presented ($T_{\text{revoke}} = \infty$ if no revocation happens prior to T_{end}). Then the contract assigns to $\mathcal{P}$ a reward of $f(\mathsf{reward}, t)$, where $t = \min(T_{\text{end}}, T_{\text{revoke}}) - T_{\text{claim}}$.

We do not explore choices of f here. We note, however, that given that a CA key $\mathsf{sk}_{\mathcal{V}}$ can be used to forge certificates for rapid use in, e.g., malware or falsified software updates, much of its value can be realized in a short interval of time which we denote by δ. (A slant toward up-front realization of the value of exploits is common in general [90].) A suitable choice of reward function should be front-loaded and rapidly decaying. A natural, simple choice with this property is

$$f(\$\mathsf{reward}, t) = \begin{cases} 0 & : t < \delta \\ \$\mathsf{reward}(1 - ae^{-b(t-\delta)}) & : t \geq \delta \end{cases}$$

for $a < 1/2$ and some positive real value b. Note that a majority of the reward is paid provided that $t \geq \delta$.

Thwarting rushing attacks. To thwart rushing attacks, we separate the claim into two phases. In the first phase, $\mathcal{P}$ expresses an intent to claim by submitting a commitment of the real claim message. $\mathcal{P}$ then waits for the next round to open the commitment and reveal the claim message. In real-life decentralized cryptocurrencies, $\mathcal{P}$ can potentially wait multiple block intervals before opening the commitment, to have higher confidence that the blockchain will not fork. In our formalism, one

round can correspond to one or more block intervals.

Figure 5.3 gives a key theft contract KeyTheft that thwarts revoke-and-claim and the rushing attacks.

5.3.3 Target and state exposure

An undesirable property of KeyTheft-Naive is that its target / victim and state are publicly visible. $\mathcal{V}$ can thus learn whether it is the target of KeyTheft-Naive. $\mathcal{V}$ also observes successful claims—i.e., whether $\mathsf{sk}_{\mathcal{V}}$ has been stolen—and can thus take informed defensive action. For example, as key revocation is expensive and time-consuming, $\mathcal{V}$ might wait until a successful claim occurs and only then perform a revoke-and-claim attack.

To limit target and state exposure, we note two possible enhancements to KeyTheft. The first is a *multi-target* contract, in which key theft is requested for any one of a set of multiple victims. The second is what we call *cover claims*, false claims that conceal any true claim. Our implementation of KeyTheft, as specified in Figure 5.3, is a multi-target contract, as this technique provides both partial target and partial state concealment.

Multi-target contract. A multi-target contract solicits the private key of any of m potential victims $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_m$. There are many settings in which the private keys of different victims are of similar value. For example, a multi-target contract KeyTheft could offer a reward for the private key $\mathsf{sk}_{\mathcal{V}}$ of *any* CA able to issue SSL/TLS certificates trusted by, e.g., Internet Explorer (of which there are more than

650 [91]).

A challenge here is that the contract state is public, thus the contract must be able to verify the proof for a valid claim (private key) $\mathsf{sk}_{\mathcal{V}_i}$ *without knowing which key was furnished, i.e., without learning* i. Our implementation shows that constructing such proofs as zk-SNARKs is practical. (The contractor $\mathcal{C}$ itself can easily learn i by decrypting $\mathsf{sk}_{\mathcal{V}_i}$, generating $\mathsf{pk}_{\mathcal{V}_i}$, and identifying the corresponding victim.)

Cover claims. As the state of a contract is publicly visible, a victim $\mathcal{V}$ learns whether or not a successful claim has been submitted to KeyTheft-Naive. This is particularly problematic in the case of single-target contracts.

Rather than sending the NIZK proof π with ct, it is possible instead to delay submission of π (and payment of the reward) until T_{end}. (That is, Claim takes as input ("claim", ct).) This approach conceals the validity of ct. Note that even without π, $\mathcal{C}$ can still make use of ct.

A contract that supports such concealment can also support an idea that we refer to as *cover claims*. A cover claim is an *invalid* claim of the form ("claim", ct), i.e., one in which ct is not a valid encryption of $\mathsf{sk}_{\mathcal{V}}$. Cover claims may be submitted by $\mathcal{C}$ to conceal the true state of the contract. So that $\mathcal{C}$ need not interact with the contract after creation, the contract could parcel out small rewards at time T_{end} to third parties that submit cover claims. We do not implement cover claims in our version of KeyTheft nor include them in Figure 5.3.

Contract $\mathsf{KeyTheft}$

Init: Set $\mathsf{state} := \textsc{init}$. Let $\mathsf{crs} := \mathsf{KeyGen}_{\mathrm{nizk}}(1^\lambda)$ denote a hard-coded NIZK common reference string generated during a trusted setup process.

Create: Same as in Contract $\mathsf{KeyTheft\text{-}Naive}$ (Figure 5.2), except that an additional parameter ΔT is additionally submitted by $\mathcal{C}$.

Intent: Upon receiving ("intent", cm) from some purported perpetrator $\mathcal{P}$:
- Assert $\mathsf{state} = \textsc{created}$
- Assert that $\mathcal{P}$ has not sent "intent" earlier
- Store $\mathsf{cm}, \mathcal{P}$

Claim: Upon receiving ("claim", ct, π, r) from $\mathcal{P}$:
- Assert $\mathsf{state} = \textsc{created}$
- Assert $\mathcal{P}$ submitted ("intent", cm) earlier such that $\mathsf{cm} = \mathsf{comm}(\mathsf{ct}||\pi, r)$.

Continue in the same manner as in contract $\mathsf{KeyTheft\text{-}Naive}$, except that the ledger update $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \$\mathsf{reward}$ does not take place immediately.

Revoke: On receive ("revoke", Π_{revoke}) from some $\mathcal{R}$:
- Assert Π_{revoke} is valid, and $\mathsf{state} \neq \textsc{aborted}$.
- $\mathsf{ledger}[\mathcal{R}] := \mathsf{ledger}[\mathcal{R}] + \$\mathsf{smallreward}$.
- If $\mathsf{state} = \textsc{claimed}$:
  - Let $t :=$ (time elapsed since successful **Claim**).
  - Let $\mathcal{P} :=$ (successful claimer).
  - $\mathsf{reward}_{\mathcal{P}} := f(\$\mathsf{reward}, t)$.
  - $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \mathsf{reward}_{\mathcal{P}}$.
- Else, $\mathsf{reward}_{\mathcal{P}} := 0$
- $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \$\mathsf{reward} - \$\mathsf{smallreward} - \mathsf{reward}_{\mathcal{P}}$
- Set $\mathsf{state} := \textsc{aborted}$.

Timer: If $\mathsf{state} = \textsc{claimed}$ and at least ΔT time elapsed since **Claim**:
- $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \$\mathsf{reward}$;
- Set $\mathsf{state} := \textsc{aborted}$.

Else if current time $T > T_{\mathrm{end}}$ and $\mathsf{state} \neq \textsc{aborted}$:
- $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \$\mathsf{reward}$.
- Set $\mathsf{state} := \textsc{aborted}$.

// $\mathcal{P}$ should not submit claims after $T_{end} - \Delta T$.

Figure 5.3: Key compromise CSC that thwarts the revoke-and-claim attack and the rushing attack.

1-Target	*#threads*	**RSA-2048**	**ECDSA_P256**
Key Gen.[$\mathcal{C}$]	1	124.88 sec	242.30 sec
	4	33.53 sec	73.38 sec
Eval. Key		215.93 MB	448.24 MB
Ver. Key		6.09 KB	5.15 KB
Prove[$\mathcal{P}$]	1	41.02 sec	83.63 sec
	4	**15.7 sec**	**32.19 sec**
Proof		711 B	711 B
Verification [Contract]		**0.0089 sec**	**0.0087 sec**

500-Target	*#threads*	**RSA-2048**	**ECDSA_P256**
Key Gen.[$\mathcal{C}$]	1	161.56 sec	263.07 sec
	4	43.35 sec	78.31 sec
Eval. Key		279.41 MB	490.85 MB
Ver. Key		4.99 KB	4.99 KB
Prove[$\mathcal{P}$]	1	54.15 sec	84.69 sec
	4	**23.54 sec**	**33.49 sec**
Proof		711 B	711 B
Verification [Contract]		**0.0087 sec**	**0.0087 sec**

Table 5.1: Performance of the key-compromise zk-SNARK circuit for **Claim** in the case of a 1-target and 500-target contracts. [.] refers to the entity performing the computational work.

5.3.4 Implementation

We rely on zk-SNARKs for efficient realization of the protocols above. As mentioned earlier, zk-SNARKs have weaker security than what is needed in UC-style simulation proofs. We therefore use a generic transformation [16] to lift security such that the zero-knowledge proof ensures *simulation-extractable soundness*. (Recall that a one-time key generation phase is needed to generate two keys: a public evaluation key, and a public verification key. To prove a certain NP statement, an untrusted prover uses the evaluation key to compute a succinct proof; any verifier can use the public verification key to verify the proof. The verifier in our case is

the contract.) In our implementation, we assume the key generation is executed confidentially by a trusted party; otherwise a prover can produce a valid proof for a false statement. To minimize trust in the key generation phase, secure multi-party computation techniques can be used as in [92].

zk-SNARK circuits for Claim. To estimate the proof computation and verification costs required for **Claim**, we implemented the above protocol for theft of RSA-2048 and ECDSA_P256 keys, which are widely used in SSL/TLS certificates currently. The circuit has two main sub-circuits: a key-check circuit, and an encryption circuit. The circuit also has other commitment and encryption sub-circuits needed for simulation sound extractability [16]. The encryption circuit was realized using RSAES-OAEP [93] with a 2048-bit key. Relying on compilers (prior to our later work on xJsnark) for high-level implementation of these algorithms was going to produce expensive circuits for the zk-SNARK proof computation. Instead, we had to build customized circuit generators that produce more efficient circuits. We then used the state-of-the-art zk-SNARK library [5] to obtain the evaluation results. Table 5.1 shows the results of the evaluation of the circuits for both single-target and multi-target contracts. The experiments were conducted on an Amazon EC2 r3.2xlarge instance with 61GB of memory and 2.5 GHz processors.

The results yield two interesting observations: i) Once a perpetrator obtains the secret key of a TLS public key, computing the zk-SNARK proof would require less than two minutes, costing less than 1 USD [76] for either single or multi-target contracts; ii) The overhead introduced by using a multi-target contract with 500

keys on the prover's side is only 13 seconds in the worst case. In the same time, the verification overhead by the contract is still the same as in the single-target case. This is achieved by the use of an efficient Merkle tree circuit that proves the membership of the compromised public key in the target key set in zero knowledge, while using the same components of the single-target circuit as is.

Validation of revoked certificates. The reward function in the contract above relies on certificate revocation time, and therefore the contract needs modules that can process certificate revocation proofs, such as CRLs and OCSP responses, and verify the CA digital signatures on them. As an example, we measured the running time of `openssl verify -crl_check` command, testing the revoked certificate at [94] and the CRL last updated at [95] on Feb 15th, 2016, that had a size of 143KB. On average, the verification executed in about 0.016 seconds on a 2.3 GHz i7 processor. The signature algorithm was SHA-256 with RSA encryption, with a 2048-bit key. Since OCSP responses can be smaller than CRLs, the verification time could be even less for OCSP.

The case of multi-target contracts. Verifying the revocation proof for single-target contracts is straightforward: The contract can determine whether a revocation proof corresponds to the targeted key. In multi-target contracts, though, the contract does not know which target key corresponds to the proof of key theft $\mathcal{P}$ submitted. Thus, a proof is needed that the revocation corresponds to the stolen key, and it must be submitted by $\mathcal{C}$.

We built a zk-SNARK circuit through which $\mathcal{C}$ can prove the connection be-

tween the ciphertext submitted by the perpetrator and the compromised target key. For efficiency, we eliminated the need for the key-check sub-circuit in **Revoke** by forcing $\mathcal{P}$ to append the secret index of the compromised public key to the secret key before applying encryption in **Claim**. The evaluation in Table 5.2 illustrates the efficiency of the verification done by the contract receiving the proof, and the practicality for $\mathcal{C}$ of constructing the proof. In contrast to the case for **Claim**, the one-time key generation for this circuit must be done independently from $\mathcal{C}$, so that $\mathcal{C}$ cannot cheat the contract. We note that the **Revoke** circuit we built is invariant to the cryptosystem of the target keys.

Table 5.2: Performance of the key-compromise zk-SNARK circuit for **Revoke** needed in the case of multi-target contract. [.] refers to the entity performing the computational work.

	#threads	**RSA-2048**	**ECDSA_P256**
Key Gen.	1	124.64 sec	124.35 sec
	4	33.52 sec	33.38 sec
Eval. Key		215.41 MB	214.81 MB
Ver. Key		5.51 KB	4.88 KB
Prove[$\mathcal{C}$]	1	41.08 sec	40.96 sec
	4	**15.94 sec**	**15.59 sec**
Proof		711 B	711 B
Verification [Contract]		**0.0087 sec**	**0.0086 sec**

5.4 Countermeasures

The main aim of our work is to emphasize the importance of developing countermeasures against CSCs for emerging smart contract systems such as Ethereum. We briefly discuss this challenge here.

Ideas such as blacklisting "tainted" coins / transactions—those with known criminal use—have been brought forward for cryptocurrencies such as Bitcoin. A proactive alternative is an identity-escrow idea in early (centralized) e-cash systems sometimes referred as "trustee-based tracing" [96, 97]. Trustee-tracing schemes permitted a trusted party ("trustee") or a quorum of such parties to trace monetary transactions that would otherwise remain anonymous. In decentralized cryptocurrencies, however, users do not register identities with authorities—and many would object to doing so. It would be possible for users to register voluntarily and to choose only to accept only currency they deem suitably registered. The idea of tainting coins, though, has been poorly received by the cryptocurrency community because it undermines the basic cash-like property of fungibility [98, 99], and trustee-based tracing would have a similar drawback. It is also unclear what entities should be granted the authority to perform blacklisting or register users.

We observe, however, that it is economically advantageous for most users of a cryptocurrency to monitor and/or restrain criminal activity, which can degrade acceptance and therefore market value. This observation has stimulated the creation, for instance, of the Blockchain Alliance [100], whose mission is to combat criminal activity on blockchains. Similarly, core developers of cryptocurrencies such as Ethereum have indicated the desirability of filtering blockchain content, by analogy with censorship of hate speech [101].

Identifying Bitcoin transactions as criminal is challenging, as transactions themselves carry no information about payment context. In contrast, CSCs, if identified as such (e.g., an assassination contract), are self-incriminating objects.

Reversing CSC binaries could be challenging, but we note that for CSCs to be effective—as in our examples above—their deployers must advertise, drawing attention to the nature of their contracts. (For example, a contractor looking to have an assassination performed must find an assassin.) Our hypothesis, therefore, is that a sufficiently vigilant cryptocurrency community can detect the presence of many CSCs and will be incentivized to filter or purge associated transactions. One simple potential mechanism is for miners to omit transactions from blocks when they are flagged by reputable communities as CSCs.

A more aggressive approach is possible as well, a notion that we call *trustee-neutralizable smart contracts*. A smart contract system might be designed such that an authority, quorum of authorities, or suitable set of general system participants is empowered to remove a contract from the blockchain. Such an approach would have a big advantage over traditional trustee-based protections, in that it *would not require users to register identities*. Whether the idea would be palatable to cryptocurrency communities and whether a broadly acceptable set of authorities could be identified are, of course, open questions, as are the right supporting technical mechanisms.

In general, our work here is therefore important in *sensitizing the cryptocurrency community to the threat of CSCs*, enabling structures for monitoring and appropriate countermeasures to be set in place. We believe it is important to create awareness in the early stages of development of decentralized smart contract ecosystems such as Ethereum.

5.5 Conclusion

We have demonstrated that a range of commission-fair *criminal smart contracts* (CSCs) are practical in decentralized currencies with smart contracts. In this chapter, we presented two crimes—leakage of secrets, and key theft crimes—and showed that they are efficiently implementable with existing cryptographic techniques, given suitable support in smart contract systems such as Ethereum. KeyTheft would require only modest, already envisioned opcode support for zk-SNARKs for efficient deployment.

We emphasize that smart contracts in distributed cryptocurrencies have numerous promising, legitimate applications and that banning smart contracts would be neither sensible nor, in all likelihood, possible. The urgent open question raised by our work is thus how to create safeguards against the most dangerous abuses of such smart contracts while supporting their many beneficial applications.

Chapter 6: Conclusions and Future Directions

6.1 Conclusions

In this thesis, we presented tools and protocols for verifiable computations. We presented a high-level framework xJsnark which makes it easier for programmers to develop secure and efficient circuits, while enabling various new optimizations for arithmetic circuits for zk-SNARKs. To the best of our knowledge, our work yields the most optimized circuits for cryptographic primitives like AES and RSA, in addition to being able to perform the optimizations automatically while compiling high-level code. We illustrated the ability of xJsnark for compiling multiple cryptographic and random memory access applications to efficient circuits, including the ZeroCash circuit [7].

We also presented practical protocols that utilize verifiable computation in cryptocurrency applications. We investigated both the good side, by designing HAWK a system for privacy-preserving smart contracts, and the dark side by investigating misuses using smart contracts and verifiable computation.

6.2 Future Directions and Ongoing work

6.2.1 Beyond Cryptocurrencies

The protocols presented in this thesis focused on cryptocurrency applications. Extending VC techniques to support other application domains such as medical or machine learning application will be beneficial. Below, we summarize one direction that we have made some progress towards, and in Section 6.2.2 we summarize related open problems.

6.2.1.1 Verifiable Research

Recently, many cases of research misconduct were discovered and several papers have been retracted [102], including cases where some scientists fabricate data, falsify results, or use questionable research methods. In one survey in 2009, 2% of scientists admitted to falsifying studies at least once and 14% admitted to personally witnessing a fabrication [103]. According to the results, misconducts were reported more in medical research. Another form of misconduct is questionable research practices (QRPs) [104], which involves practices in which researchers intentionally make their results look better by omitting data, or handling outliers in certain ways for example. These were found to be more prevalent among researchers [103, 104]. Furthermore, looking to the cases that were discovered in data fraud and falsifications, the reason for the discovery of such cases was how anomalous the faked results were, which means that some cases may be more difficult to discover if the results

look within the normal and acceptable possible range, which one blog post called it the *perfect scientific crime* [105]. This motivates the need for a process that can ease the process of verification, while making it harder for researchers to fake or falsify data. Using new technologies like verifiable computation, trusted hardware and smart contracts, we make an attempt at making the research publication process more trustworthy and reliable to reduce cases like data fraud and questionable research methods (whenever possible).

These practices make the scientific publication process less reliable, and may even put patients at risk in some clinical scenarios. In this project, we aim to use efficient zero-knowledge proofs to develop a practical service that enables reviewers to verify research publications even when the data involved is private.

Our work aims to consider three domains: clinical studies where patients' samples are analyzed by a lab, surveys via an online survey service and surveys done via in-person interviews with the subjects. One of the main challenges is how to reduce or minimize trust in any intermediaries. In some cases like in-person interviews, there are usually no intermediaries, which makes the problem even more challenging.

6.2.2 Open Problems

6.2.2.1 Efficiency versus Trust Assumptions

Recently many zero-knowledge constructions have been proposed [106–110] in order to solve the drawbacks of the zk-SNARK construction that was used in this

thesis, namely the need for a trusted setup for every different computation.

However, to the best of our knowledge, the verification of the proofs produced by these schemes is not as efficient as the zk-SNARK constructions we use, and the proofs are not as succinct, which makes it hard for such new techniques to be adopted by blockchain-based applications directly.

A research question here will be how to solve the tension between efficiency and trusted setup, by finding ways to maintain both succinctness and trustless setup, or minimized trusted setup.

6.2.2.2 Large-scale Problems

Although the zk-SNARK constructions used in this thesis provide a good level of succinctness that primarily benefits the verifier, such constructions require high computation resources for very large arithmetic circuits in order to compute the proof. This could limit their applicability in domains like machine learning and image processing applications, where circuits can easily reach billions of gates. Enabling verifiable computation techniques to scale to such circuit sizes while maintaining succinctness is an open question.

Appendix A: Low-level Circuit Development Tools

In this appendix, we summarize other development tools that we developed for building efficient and secure zk-SNARK applications.

A.1 Jsnark

jsnark is a low-level circuit construction library written in Java [14]. Its main goal is to provide an easier and more friendly front end to libsnark [15]. jsnark aims at making low-level development easier by providing high-level APIs and few back end optimizations to enable programmers to write optimized programs conveniently. The library also allows developers to augment circuits produced by the Pinocchio compiler in a straightforward way. jsnark has been used in developing and investigating applications like HAWK [2] (Chapter 4) and Gyges [8] (Chapter 5), and was used as a starting library for both C∅C∅ [16] and xJsnark. It was also used for benchmarking for several other works.

A.2 C∅C∅

C∅C∅ is a library for developing composable zero-knowledge proofs [16]. It also provides a library for SNARK-friendly cryptographic primitives which enables imple-

menting composable zero-knowledge proofs efficiently. More details about SNARK-friendly cryptographic primitives are provided in Appendix B.

As we mention in Section 4.4.1, known instantiations of zk-SNARKs [11,20,25] are not known to satisfy composability and therefore often cannot be adopted straight out of the box in the design of larger protocols, as UC-secure protocols would often require *simulation sound extractable* zero-knowledge proofs. For this purpose, we built C∅C∅ (short for **C**omposable **0**-knowledge, **C**ompact **0**-knowledge), which enables practical, UC-secure, non-interactive zero-knowledge proofs for *general, user-defined statements*.

C∅C∅ relies on UC-secure NIZKs that are *circuit succinct*, but not *witness succinct*. In other words, the size of the proofs and verification time are linear in the witness size, but independent of the size of the circuit that encodes the language. Note that in comparison, standard, non-UC-secure SNARKs achieve a stronger notion of succinctness, i.e., they are both circuit- and witness succinct. Note that recent progress in the area shows that it is possible to get succinct composable NIZK proofs [78].

In order to implement such proofs efficiently, this work also explores selected known cryptographic primitives such as encryption and key exchange, and expresses them as efficient arithmetic circuits. Earlier and concurrent works have considered the optimization of such primitives for specific purposes, such as [6, 7, 9, 69]. In this work, we studied possible optimizations/alternatives, including the trade-offs of existing constructions such as RSA and field extension for key exchange, and propose to use a customized elliptic curve-based construction that is more efficient.

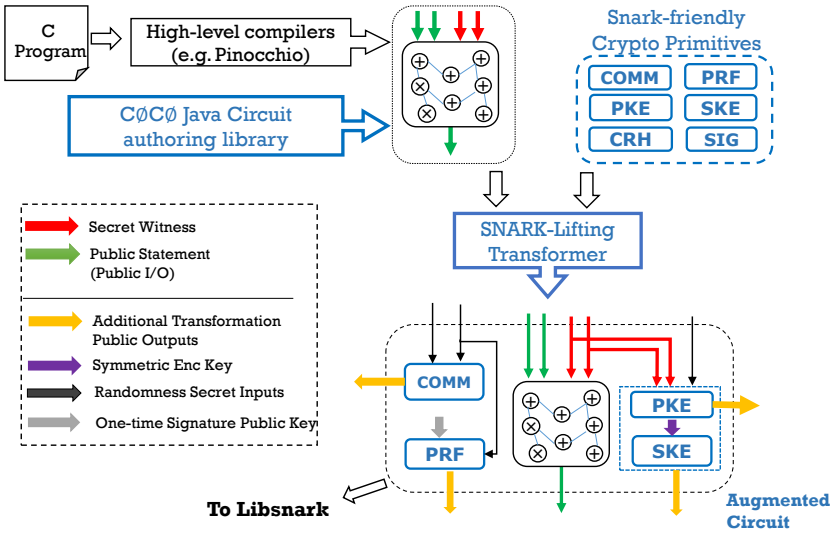

Figure A.1: CØCØ's architecture. The SNARK lifting module transforms a zk-SNARK to a simulation sound extractable NIZK (a UC-secure NIZK).

Similarly, SNARK-friendly block ciphers for symmetric encryption are explored. More details are in Appendix B. Our implementation for such schemes is available in the jsnark repository [14].

From the front end perspective, using CØCØ, the developer only will need to implement the high-level application (i.e., user-defined statement), and the CØCØ framework handles the remaining automatically and results in highly optimized implementations. Figure A.1 presents the high-level architecture of xJsnark and highlights our contributions. CØCØ has been adopted in subsequent works [2, 8] to build privacy-preserving protocols. Releasing CØCØ to the public is in progress.

Appendix B: SNARK-friendly Cryptographic Primitives

In this section, we explore alternatives for different cryptographic primitives. As discussed earlier, known SNARK constructions model computation as arithmetic circuits modulo a large prime p. Standard implementations and parameter choices for cryptographic primitives are targeted at modern hardware platforms with different constraints than SNARKs. For example, some operations, like addition and constant-scalar multiplication of field elements $\mathbb{F}_p$, which are expensive in hardware, are essentially free in a SNARK; however, while XORing two 32-bit numbers takes a single cycle on an ordinary CPU, this is far more costly in an arithmetic circuit.

Guided by the cost model we described in Section 2.1.1, we explore other alternatives that could be more circuit-friendly when represented as SNARK circuits. This was primarily motivated by the need to transform zk-SNARKs to simulation sound extractable NIZKs, a property need for UC-based proofs. We call this transformation: SNARK lifting.

B.1 Encryption

As the costs of all SNARK-lifting constructions that we use are dominated by the public-key encryption of the witness [16], we focus most of our efforts on

this task. We start by implementing an optimized circuit for RSA PKCS1 v1.5 and RSA-OAEP (PKCS1 v2.2) encryption with SHA-256 as the hash and mask generation function [93], as our baselines. We apply our xJsnark optimizations here, although they were not available at the time of the initial version of this work [16].

We consider the use of hybrid encryption, where we use a public-key scheme to exchange/encrypt a symmetric key, and then use the symmetric key to encrypt the plaintext. To explore this alternative, we explored various options for key exchange and symmetric-key encryption as we describe below.

Key exchange. Although it is possible to use the optimized RSA circuit above to encrypt and exchange a symmetric key, we explored the following schemes for a SNARK-friendly key exchange in order to find a more efficient alternative than RSA.

- *Diffie-Hellman key exchange via a SNARK-friendly field extension.* Instead of relying on RSA for key exchange, we investigate another scheme based on the Discrete-Logarithm (DL) problem in Extension Fields, and use it for symmetric key exchange. Since p is only 254-bit prime in existing implementations, the DL problem in $\mathbb{F}_p$ will not be hard, therefore an extension $\mathbb{F}_{p^\mu}$ will be used instead. This idea is mainly inspired by the construction in Pinocchio-Coin [6]. The key exchange circuit has two generators in that case $g, h \in \mathbb{F}_{p^\mu}$, where $\langle g \rangle = \langle h \rangle$ is a large multiplicative subgroup of order $q | p^\mu - 1$. We follow Lentra's guidelines for selecting q to be a factor of the μ-th cyclotomic polynomial $\Phi_\mu(x)$ when evaluated at $x = p$ [111].

The hardness of discrete-log in extension fields has been studied for quite some time; recently quasi-polynomial time algorithms [112, 113] have been designed for the special case of fixed-sized, i.e., small, characteristic fields. The key ideas behind these recent algorithms, however, do not extend to larger characteristic fields. To estimate the security level in our case, we observe that the finite field in our context is related to pairing-based curves due to the underlying implementation of SNARKs using BN curves [114]. In fact, the prime p has a special form in our case (p can be computed based on a polynomial $36x^4 + 36x^3 + 18x^2 + 6x + 1$). This property can be utilized to solve the Discrete-Logarithm problem faster using the the Special Number Field Sieve algorithm proposed by Joux et al [115]. This is the best attack (see [116]) we are aware of; using $\mu = 4$ in our scenario yields about 86-bit security.

The extension field construction requires us to search for large primes that divide $\Phi_\mu(p)$. In our implementation using libsnark [5], in order to get about 80-bit level of security, we set μ to be 4 as mentioned above, and choose q to be the 398-bit prime factor of the $\Phi_4(p)$, where p is the SNARK field order of libsnark. For higher security when $\mu = 6$, we found a 313-bit prime order subgroup for the extension field. However, to get higher security levels (i.e., $\mu > 6$), this may require expensive factorization.

- *Diffie-Hellman key exchange via a SNARK-friendly Elliptic Curve.* The field extension approach above has two drawbacks: 1) The size of public keys and keying material is large. For 80-bit security, the size of the exchanged key

is nearly 128 bytes. A hash-based key derivation function (KDF) will have a high cost for SNARKs, especially if we raise the security level of the field extension to above 100. 2) It requires expensive factorization to find suitable parameters to achieve higher bit security. Therefore, we investigated whether we can construct a SNARK-friendly elliptic curve mainly for key exchange. Note that in earlier works that proposed elliptic curves for SNARKs [12, 69], one goal was to implement the pairing operation efficiently within the circuits. On the other hand, our goal here to implement the operation required in key exchange, i.e. g^x, in a more efficient way.

Following the guidelines described in constructing Curve25519 [117], we propose a SNARK-friendly Montgomery elliptic curve over the SNARK field $\mathbb{F}_p$, that is specified by the equation: $y^2 = x^3 + Ax^2 + x$, where $A = 126932$.

Choosing $A = 126932$ implies that the order of the curve is $8\times$ 251-bit prime, and the order of its twist is $4\times$ 252-bit prime. Note that the size of the prime order subgroup is above 2^{250}, achieving about 125-bit security. The secret key in our construction has the same properties as in Curve25519, i.e. chosen to be a multiple of 8, in order to avoid small subgroup attacks. Note that we don't follow the other efficiency guidelines described for Curve25519, due to the different setting and cost model. Finally, the safety of the parameters of the new curve was verified according to the script available online on the Safe Curves website [118].

The implementation of point addition and multiplication using the above curve

is very efficient. Assuming affine coordinates, both point doubling and addition can both be encoded using the same number of constraints in our case. In fact, each step only requires 4 multiplication gates. This is because verifying multiplicative inverses in SNARKs is very cheap, costing one multiplication gate per operation. Furthermore, to implement the operation g^x efficiently, one possible optimization is to pre-compute all powers for a base g, and hardcode them in the circuit, i.e. $\{g^2, g^3, .., g^{251}\}$, then use them to compute g^x. In our implementation, the operation g^x costs 6 multiplications per each bit in x (compared to 60 multiplications in Geppetto's curve [12]).

Symmetric-key encryption. After exchanging a secret key using a public key scheme, symmetric encryption is performed in CBC mode using a block cipher. Note that in the case of using the extension field-based scheme or the elliptic curve scheme, we use a hash-based key derivation function to derive a secret key and a secret initialization vector for the symmetric encryption. This process is not required in the RSA case, as the sender can encrypt the random key and initialization vector directly.

To achieve better performance, we looked for lightweight ciphers according to the criteria in Section 2.1.1, and found two promising ciphers, Speck and Chaskey block ciphers. Speck was proposed in 2013 [72] by NSA. The Chaskey block cipher is an Even-Mansour block cipher that was used in the Chaskey MAC algorithm [73]. In terms of security, this block cipher relies on a weaker security model, in which the time complexity of an attack is about $2^k/D$ if the attacker obtains D chosen

plaintext-ciphertext pairs, and the key size is k bits. For increased security, we use a more secure version of the Chaskey block cipher called Chaskey-LTS, which applies 16 rounds instead of 8 to achieve long-term security. Both Speck and Chaskey-LTS ciphers have more SNARK-friendly implementations compared to AES, but the disadvantage of using these ciphers is that they are new compared to AES, and not as trusted. This has motivated us to later research the optimization for AES presented in xJsnark (Chapter 3).

Micro-benchmarks. Table B.1 provides the micro-benchmarks for the public key and symmetric key schemes discussed above. It should be noted that for public key schemes, we assumed that the public key is hardcoded in the circuit, which is suitable for our purposes in the SNARK lifting transformations. If the public keys are not hardcoded, the cost for the field extension circuit will increase with about 20k gates, and the cost for the elliptic curve scheme will be about 5K gates, but it will result into minor difference in the RSA case. As noted in the table, the cost of SNARK-friendly elliptic curve is more than 3-10x better than the RSA-2048 case. However, the cost of Field Extension is worse than the RSA-1024 v1.5 case, but better than the OAEP case. Note that it costs more gates compared to the elliptic curve setting due to the cost of hash-based key derivation using SHA-256 (Exploring SNARK-friendly key derivation functions can be a direction for future work). For the block ciphers, the table shows about 2x better cost for Speck and Chaskey compared to an optimized version of AES.

Table B.1: Number of constraints of public-key and symmetric-key encryption. The field extension uses ($\mu = 4$). The block cipher schemes all use a 128-bit key. The block cipher cost does not include any one-time key expansion cost.

PKE Scheme	Key Exchange + Derivation	**Block Cipher**	Cost / Block
RSA-PKCS1 v1.5 (1024)	46k + 0	AES	12.3k
RSA-OAEP (1024)	165k + 0	Speck	6.5k
Field Extension	3.5k + 52k	Chaskey	5k
RSA-PKCS1 v1.5 (2048)	94k + 0		
RSA-OAEP (2048)	348k + 0		
Elliptic Curve	3k + 26k		

B.2 Other Cryptographic Primitives

PRFs and commitments. In our implementation, we instantiate PRFs and Commitments using an efficient SHA-256 circuit. An efficient SHA-256 circuit costs about 26k gates for one block (512-bit input), while its naïve implementation using SNARK compilers costs more than 40k gates. The optimizations are mainly achieved by representing Boolean operations efficiently, and careful circuit design. A previous similar implementation and a detailed discussion of SHA-256 optimizations can be found in [7].

Collision resistant hashes. Lattice-based cryptography, including Ajtai's collision resistant hash, are promising for use in SNARKs [69]. However, existing estimates of concrete security for such schemes only extend to lattices over small finite fields, but do not *a priori* apply to lattices constructed over a SNARK's (much larger) native field. In [16], parameterization of such lattice-based schemes is pro-

vided, which we use for collision resistant hashing schemes in our circuits.

Signatures. For digital signatures, one possible approach is to use an optimized RSA-PSS signature verification circuit using the PKCS-1 standard v2.1 [93]. As stated earlier, SNARK circuits do not necessarily have to compute, and since the signature verification in RSA is cheaper (due to the small public exponent), in our implementations involving digital signatures, we adopt a signature verification circuit instead, and apply the same optimizations we applied for the RSA Encryption circuit. We currently use SHA-256 to hash the message to be signed. Another optimization that can be done is to use the same approach we used for optimized DH key exchange, by relying on a SNARK-friendly elliptic curve-based scheme for signatures. Such scheme can actually be derived based on the curve used earlier in key exchange, like the relation between the curve used in Ed25519 [119] and Curve25519 [117]. Since the protocols we implemented in this thesis do not rely mainly on signatures within the circuits, we leave investigating this direction to future work.

Appendix C: Additional Illustrative Examples

C.1 xJsnark

C.1.1 Sorting via Permutation Verifier

This example illustrates the usage of both the permutation verifier feature along with the external code blocks. The omitted code is simple Java sorting calls. Writing code using this approach leads to performance that is order of magnitudes better than just writing merge sort code, as benchmarked in earlier work.

```
Program Sort {
  int SIZE = 1024;
  uint_32 [] array = new uint_32[SIZE];
  uint_32 [] sortedArray = new uint_32[SIZE];

  inputs { array };
  witnesses { sortedArray };
  outputs { sortedArray };

  void Main(){
    external {
        // outside circuit

        // extract values
        BigInteger [] values = new BigInteger[SIZE];
        for (int i = 0; i < SIZE; i++)
          values[i] = array[i].val;

    /** code omitted ..
        Apply sorting outside the circuit to obtain sortedValues
         and sortedIdx(the index of elements after sorting). **/

        // provide solution
        for (int i = 0; i < SIZE; i++)
```

```
            sortedArray[i].val = sortedValues[i];

        // Give hint to the evaluator during run time
        resolve_permutation (sortedIdx, "id1");
    }
    // Inside circuit
    verify_permutation <uint_32>( array, sortedArray, "id1" );
    for (int i = 0; i < SIZE - 1; i++)
      verify ( sortedArray[i] <= sortedArray[i + 1] );
  }
}
```

C.1.2 ZeroCash

As a case study, we illustrate how the framework can be used to program the pour circuit of ZeroCash [7], and compare the resulting number of gates, and the implementation effort with the manual implementation.

The Pour circuit is the main circuit used in ZeroCash to hide the flow of money, relying on the power of zk-SNARKs. The circuit relies mainly on SHA-256 as a building block, and uses it to instantiate commitments, Merkle trees and PRFs. In the original ZeroCash paper [7], the benchmarks assumed 2^{64} total number of coins, which lead to a large circuit consisting of millions of gates. This made the manual implementation more attractive in comparison with the existing high-level compilers at the time. The ZeroCash paper reported the number of multiplication gates to be 4109330, after **manual optimization**. (A slightly better implementation is available here [62], achieving about 4017157 gates).

In comparison with the above low-level implementation, our framework achieves a very close and actually better number due to some low-level arithmetic optimizations that can be automatically detected by the multi-variate polynomial minimizer.

Our framework provides a total of **3814264** gates, saving more than 2×10^5 gates. This is while the programmer provided the code in a high-level manner, without specifying any of the low-level optimizations.

In the following subsections, we list the code written for the Pour circuit using xJsnark. Note that this is **all** the code the programmer writes in our case. There are no specific manual optimizations done by the programmer in any of the code snippets. The code tries to follow the namings used in the original ZeroCash paper [7] for better readability (The reader may consult Figure 2 in the ZeroCash Paper for the detailed specifications).

C.1.2.1 ZeroCash Data Structures

1. Coin Information. Every coin structure includes a secret value, randomness secrets, and a public address.

```
struct Coin {
  uint_64 value;
  uint_32 [] rho = new uint_32[8];
  uint_32 [] rand = new uint_32[12];
  PubKey pubKey = new PubKey();
}
```

The following data structures store the key information. Note that the keys in ZeroCash also include encryption keys but they are not part of the circuit.

```
struct PrivKey {
  uint_32 [] a_sk = new uint_32[8];
}

struct PubKey {
  Digest a_pk = new Digest();
}
```

2. Hash Digests. For readability, this is a simple data structure to represent a

SHA-256 output. It also includes a method for equality assertion.

```
struct Digest {

  uint_32[] array = new uint_32[8]; // Typically SHA-256 output

  void assertEqual(Digest other) {
    for (int i = 0; i < array.length; i++) {
      verifyEq (array[i], other.array[i]); // built-in equality assertion
    }
  }
}
```

3. Merkle Tree Authentication Path. This data structure represents the Merkle tree authentication path. It includes an integer to specify to which direction, the witness digests are added (left or right). This structure also defines a method to compute the merkle tree root. As will be shown later, the programmer instantiates one or more MerkleAuthPath objects, labels them as witnesses, and verifies the root computed through the Merkle tree.

```
struct MerkleAuthPath {

  Digest[] digests = new Digest[PourCircuit.HEIGHT];
  uint_64 directionSelector; // Path specification


  public MerkleAuthPath() {
    for (int i = 0; i < digests.length; i++) {
      digests[i] = new Digest();
    }
  }

  Digest computeMerkleRoot(Digest leaf) {
    bit[] directionBits = directionSelector.bits;
    Digest currentDigest = leaf;

    uint_32[] inputToNextHash = new uint_32[16];

    for (int i = 0; i < PourCircuit.HEIGHT; i++) {
      for (int j = 0; j < 16; j++) {
        if (directionBits[i]) {
          inputToNextHash[j] = j >= 8 ? currentDigest.array[j - 8] : digests[i].array[j];
        } else {
          inputToNextHash[j] = j < 8 ? currentDigest.array[j] : digests[i].array[j - 8];
        }
      }
```

```
      currentDigest = Util.SHA256(inputToNextHash);
    }
    return currentDigest;
  }
}
```

C.1.2.2 Utilities

This mainly includes the code for the SHA-256 hash function. Its code is pretty standard without any SNARK hints that could benefit the framework's backend.

```
public class Util {

  public static Digest SHA256(uint_32[] input) {

    uint_32 [] K = {/** SHA-256 Hardcoded Constants **/}
    uint_32 [] H = {/** SHA-256 Hardcoded Constants **/};

    uint_32 [] words = new uint_32[64];
    uint_32 a = H[0];
    uint_32 b = H[1];
    uint_32 c = H[2];
    uint_32 d = H[3];
    uint_32 e = H[4];
    uint_32 f = H[5];
    uint_32 g = H[6];
    uint_32 h = H[7];

    for (int j = 0; j < 16; j++) {
      words[j] = input[j];
    }

    for (int j = 16; j < 64; j++) {
      uint_32 s0 = rotateRight(words[j - 15], 7) ^ rotateRight(words[j - 15], 18) ^ (words[j
    - 15] >> 3);
      uint_32 s1 = rotateRight(words[j - 2], 17) ^ rotateRight(words[j - 2], 19) ^ (words[j -
    2] >> 10);
      words[j] = words[j - 16] + s0 + words[j - 7] + s1;
    }

    for (int j = 0; j < 64; j++) {
      uint_32 s0 = rotateRight(a, 2) ^ rotateRight(a, 13) ^ rotateRight(a, 22);
      uint_32 maj = (a & b) ^ (a & c) ^ (b & c);
      uint_32 t2 = s0 + maj;
      uint_32 s1 = rotateRight(e, 6) ^ rotateRight(e, 11) ^ rotateRight(e, 25);
      uint_32 ch = (e & f) ^ (~(e) & g);
      uint_32 t1 = h + s1 + ch + K[j] + words[j];
```

```
      h = g;
      g = f;
      f = e;
      e = d + t1;
      d = c;
      c = b;
      b = a;
      a = t1 + t2;
    }

    H[0] = H[0] + a;
    H[1] = H[1] + b;
    H[2] = H[2] + c;
    H[3] = H[3] + d;
    H[4] = H[4] + e;
    H[5] = H[5] + f;
    H[6] = H[6] + g;
    H[7] = H[7] + h;

    Digest out = new Digest();
    out.array = H;
    return out;
  }

  public static uint_32 rotateRight(uint_32 in, int r) {
    return (in >> r) | (in << (32 - r));
  }


  public static uint_32[] concat(uint_32[] a1, int idx1, int l1, uint_32[] a2, int idx2, int
      l2) {
    uint_32[] res = new uint_32[l1 + l2];
    for (int i = 0; i < l1; i++) {
      res[i] = a1[i + idx1];
    }
    for (int i = 0; i < l2; i++) {
      res[i + l1] = a2[i + idx2];
    }
    return res;
  }
}
```

C.1.2.3 The ZeroCash Pour Circuit

The following represents the program representing the Pour circuit written using xJsnark.

```
Program PourCircuit {
```

```
 /** Merkle tree height **/
 public static final int HEIGHT = 64;

 /** Merkle tree root **/
 Digest root = new Digest();


 /** Merkle tree authentication paths **/
 MerkleAuthPath authPath1 = new MerkleAuthPath();
 MerkleAuthPath authPath2 = new MerkleAuthPath();

 /** Coin Data **/
 Coin c1_old = new Coin();
 Coin c2_old = new Coin();
 Coin c1_new = new Coin();
 Coin c2_new = new Coin();

 /** Serial numbers used to prevent double spending **/
 Digest sn1_old = new Digest();
 Digest sn2_old = new Digest();

 /** Coin Commitments **/
 Digest c1_old_comm = new Digest();
 Digest c2_old_comm = new Digest();
 Digest c1_new_comm = new Digest();
 Digest c2_new_comm = new Digest();

 /** Secret keys of old coins **/
 PrivKey sk1_old = new PrivKey();
 PrivKey sk2_old = new PrivKey();

 /** Hash of a PK used for One-time Sig **/
 Digest h_sig = new Digest();
 /** MACs to prevent malleability **/
 Digest h1 = new Digest();
 Digest h2 = new Digest();

 /** public transaction amount (e.g. transaction fees) **/
 uint_64 pubVal;

inputs {
  root, pubVal, h_sig
}

/** Data Kept Secret **/
witnesses {
  authPath1, authPath2, c1_old, c2_old, c1_new, c2_new, c1_old_comm, c2_old_comm, sk1_old,
  sk2_old
}

outputs {
  sn1_old, sn2_old, c1_new_comm, c2_new_comm, h1, h2
}
```

```
public void Main() {

  // verifying that the commitments have appeared before on the ledger
  authPath1.computeMerkleRoot(c1_old_comm).assertEqual(root);
  authPath2.computeMerkleRoot(c2_old_comm).assertEqual(root);

  // verify the knowledge of the secret keys
  c1_old.pubKey.a_pk.assertEqual(PRF("addr", sk1_old.a_sk, new uint_32[]{0, 0, 0, 0, 0, 0,
   0, 0}));
  c2_old.pubKey.a_pk.assertEqual(PRF("addr", sk2_old.a_sk, new uint_32[]{0, 0, 0, 0, 0, 0,
   0, 0}));

  // Compute old coins serial numbers (this avoids double spending)
  sn1_old = PRF("sn", sk1_old.a_sk, c1_old.rho);
  sn2_old = PRF("sn", sk2_old.a_sk, c2_old.rho);

  // Verify old commitments and compute the new ones
  c1_old_comm.assertEqual(COMM_s(COMM_r(c1_old.rand, c1_old.pubKey.a_pk.array,
   c1_old.rho).array, c1_old.value));
  c2_old_comm.assertEqual(COMM_s(COMM_r(c2_old.rand, c2_old.pubKey.a_pk.array,
   c2_old.rho).array, c2_old.value));
  c1_new_comm = COMM_s(COMM_r(c1_new.rand, c1_new.pubKey.a_pk.array,
   c1_new.rho).array, c1_new.value);
  c2_new_comm = COMM_s(COMM_r(c2_new.rand, c2_new.pubKey.a_pk.array,
   c2_new.rho).array, c2_new.value);

  // verifying the correct flow of money
  verifyEq ( c1_old.value + c2_old.value , c1_new.value + c2_new.value + pubVal );

  // verifying there are no overflows (the positivity of the values is guaranteed by the
   backend)
  uint_65 sum = uint_65(c1_old.value) + c2_old.value;
  uint_65 mask = 0x10000000000000000u;
  verifyEq ( sum & mask , 0 );

  // Compute MACs needed for non-malleability
  // 3 bits from h_sig are truncated (SEE page 23 in
  // http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf )
  // 1 bit is truncated here, and 2 bits are already truncated later in PRF call
  uint_32 [] h_sigTruncated = truncate(h_sig.array, 1);
  h1 = PRF("pk", sk1_old.a_sk, h_sigTruncated);
  h_sigTruncated[0] = h_sigTruncated[0] | 0x80000000u;
  h2 = PRF("pk", sk2_old.a_sk, h_sigTruncated);
}

/** Parametrized PRF Function **/
private Digest PRF(String type, uint_32[] x, uint_32 [] z) {

  // truncate 2 least significant bits
  // See page 22 in
   http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf
  z = truncate(z, 2);
  uint_32 mask = 0u;
```

```
    if (type.equals("addr")) {
      mask = 0u;
    } else if (type.equals("sn")) {
       mask = 0x40000000u;
    } else if (type.equals("pk")) {
      mask = 0x80000000u;
    }

    uint_32 [] input = new uint_32[16];
    for (int i = 0; i < 16; i++) {
      if (i < 8) {
        input[i] = x[i];
      } else if (i == 8) {
        input[i] = z[i - 8] | mask;
      } else {
        input[i] = z[i - 8];
      }
    }
    return Util.SHA256(input);
}


/** Commitment_r Function **/
private Digest COMM_r(uint_32[] r, uint_32[] a_pk, uint_32 [] rho) {
    uint_32 [] input1 = Util.concat(a_pk, 0, a_pk.length, rho, 0, rho.length);
    uint_32 [] out1 = Util.SHA256(input1).array;
    uint_32 [] input2 = Util.concat(r, 0, r.length, out1, 0, out1.length / 2);
    return Util.SHA256(input2);
}

/** Commitment_s Function **/
private Digest COMM_s(uint_32[] k, uint_64 val) {
    uint_32 [] paddedVal = new uint_32[]{0, 0, 0, 0, 0, 0, uint_32((val >> 32)), uint_32(val)};
    uint_32 [] input = Util.concat(k, 0, k.length, paddedVal, 0, paddedVal.length);
    return Util.SHA256(input);
}

// truncates n least signicant bits. n is assumed to be <= 32
// This is to follow the implementation decision in (page 22):
  http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf
private uint_32 [] truncate(uint_32 [] words, int n) {

    if (n > 32 || n < 0) { throw new IllegalArgumentException("Invalid truncation
   argument"); }
    uint_32 [] t = new uint_32[words.length];
    for (int i = 0; i < words.length; i++) {
      t[i] = words[i];
    }
    t[words.length - 1] = t[words.length - 1] >> n;
    for (int i = words.length - 2; i >= 0; i--) {
      t[i + 1] = t[i + 1] | (t[i] << (32 - n));
      t[i] = t[i] >> n;
    }
    return t;
```

```
  }
}
```

In comparison with the existing low-level implementation for ZeroCash that is available in [62] and its gadget dependencies in [15], it can be observed how xJsnark saves a lot of the programming effort, while producing efficient output as illustrated in Section 3.6.3.

C.2 Criminal Smart Contracts

The section illustrates an actual smart contract for public leakage. This contract fixes two main drawbacks with the existing Darkleaks protocol (Shortcomings 1 and 2 discussed in 5.2.1). The contract mainly enables better guarantees through deposits and timeout procedures, while preventing selective withholding. The following code illustrates the contract code. The main goal of providing this code is to illustrate how fast it could be to write such contracts.

```
data leaker_address
data num_chunks
data  revealed_set_size
data T_end
data deposit
data reveal_block_number
data selected_sample []
data key_hashes []
data donations []
data sum_donations
data num_donors
data  finalized

def  init ():
  self.leaker_address  = msg.sender


# A leaker commits to the hashes of the encryption keys, and sets the announcement details
def commit( key_hashes:arr,  revealed_set_size ,  reveal_block_number, T_end,
```

```
    distribution_address):
  # Assuming a deposit of a high value from the leaker to discourage aborting
  if( msg.value >= 1000000 and msg.sender == self.leaker_address and self.deposit == 0 and
    revealed_set_size < len(key_hashes)):
    self.deposit = msg.value
    self.num_chunks = len(key_hashes)
    self.revealed_set_size = revealed_set_size
    self.T_end = T_end
    self.reveal_block_number = reveal_block_number
    i = 0
    while(i < len(key_hashes)):
      self.key_hashes[i] = key_hashes[i]
      i = i + 1
    return (0)
  else:
    return (-1)


def revealSample(sampled_keys:arr):
  # The contract computes and stores the random indices based on the previous block hash.
    The PRG is implemented using SHA3 here for simplicity.
  # The contract does not have to check for the correctness of the sampled keys. This can be
    done offline by the users.
  if( msg.sender == self.leaker_address and len(sampled_keys) == self.revealed_set_size and
    block.number == self.reveal_block_number ):
    seed = block.prevhash
    c = 0
    while(c < self.revealed_set_size ):
      if(seed < 0):
        seed = 0 - seed
      idx = seed % self.num_chunks
      # make sure idx was not selected before
      while(self.selected_sample[idx] == 1):
        seed = sha3(seed)
        if(seed < 0):
          seed = 0 - seed
        idx = seed % self.num_chunks
      self.selected_sample[idx] = 1
      seed = sha3(seed)
      c = c + 1
    return(0)
  else:
    return(-1)


def donate():
  # Users verify the shown sample offline, and interested users donate money.
  prev_donation = self.donations[msg.sender]
  if( msg.value > 0 and block.timestamp <= self.T_end and prev_donation == 0):
    self.donations[msg.sender] = msg.value
    self.num_donors = self.num_donors + 1
    self.sum_donations = self.sum_donations + msg.value
    return(0)
  else:
```

```
    return(-1)

def revealRemaining(remaining_keys:arr):
  # For the leaker to get the reward, the remaining keys have to be all revealed at once.
  # The contract will check for the consistency of the hashes and the remaining keys this
    time.
  if( msg.sender == self.leaker_address and block.timestamp <= self.T_end and
    len(remaining_keys)==self.num_chunks - self.revealed_set_size and self.finalized == 0):
    idx1 = 0
    idx2 = 0
    valid = 1
    while(valid == 1 and idx1 < len(remaining_keys)):
      while(self.selected_sample[idx2] == 1):
        idx2 = idx2+1
      key = remaining_keys[idx1]
      key_hash = self.key_hashes[idx2]
      if(not(sha3(key) == key_hash)):
        valid = 0
      idx1 = idx1+1
      idx2 = idx2+1

    if(valid == 1):
      send(self.leaker_address, self.sum_donations + self.deposit)
      self.finalized = 1
      return (0)
    else:
      return(-1)
  else:
    return(-1)


def withdraw():
  ## This is a useful module that enables users to get their donations back if the leaker
    aborted
  v = self.donations[msg.sender]
  if(block.timestamp > self.T_end and self.finalized == 0 and v > 0):
    send(msg.sender, v + self.deposit/self.num_donors)
    return (0)
  else:
    return (-1)
```

The contract above considers a leaker who announces the ownership of the leaked material (e-mails, photos, secret documents, .. etc), and reveals a random subset of the encryption keys at some point to convince users of the ownership. Interested users can then deposit donations. In order for the leaker to get the reward from the contract, **all** the rest of the keys must be provided at the same

time, before a deadline.

To ensure **incentive compatability**, the leaker is required by the contract in the beginning to deposit an amount of money, that is only retrievable if complied with the protocol. Also, for users to feel safe to deposit money, a timeout mechanism is used, such that if the leaker does not provide a response in time, the users will be able to withdraw the donations.

Bibliography

[1] Riad S Wahby, Srinath Setty, Zuocheng Ren, Andrew J Blumberg, and Michael Walfish. Efficient ram and control flow in verifiable outsourced computation. In *NDSS*, 2015.

[2] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *S & P*. IEEE, 2016.

[3] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Advances in Cryptology–EUROCRYPT 2013*, pages 626–645. Springer, 2013.

[4] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[5] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security*, 2014.

[6] George Danezis, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio coin: building zerocoin from a succinct pairing-based proof system. In *Proceedings of the First ACM workshop on Language support for privacy-enhancing technologies*, pages 27–30. ACM, 2013.

[7] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on. IEEE*. IEEE, 2014.

[8] Ari Juels, Ahmed Kosba, and Elaine Shi. The ring of gyges: Investigating the future of criminal smart contracts. Cryptology ePrint Archive, Report 2016/358, 2016. `http://eprint.iacr.org/2016/358`.

[9] Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Turning shabby x.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *S& P*, 2016.

[10] Zcash. `https://z.cash/`.

[11] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, 2015.

[12] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *S&P*, 2014.

[13] Jetbrains MPS Github. `https://github.com/JetBrains/MPS`.

[14] jsnark: A java library for building zk-snark circuits. `https://github.com/akosba/jsnark`.

[15] libsnark. `https://github.com/scipr-lab/libsnark`.

[16] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, Hubert Chan, Charalampos Papamanthou, Rafael Pass, abhi shelat, and Elaine Shi. C∅c∅: A framework for building composable zero-knowledge proofs. Cryptology ePrint Archive, Report 2015/1093, 2015. `http://eprint.iacr.org/2015/1093`.

[17] Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed Kosba, Ari Juels, and Elaine Shi. Solidus: Confidential distributed ledger transactions via pvorm. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 701–717. ACM, 2017.

[18] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[19] G. Wood. Ethereum: A secure decentralized transaction ledger. `http://gavwood.com/paper.pdf`, 2014.

[20] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *S & P*, 2013.

[21] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, 2014.

[22] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay: a secure two-party computation system. In *USENIX Security Symposium*, 2004.

[23] Benjamin Braun, Ariel J Feldman, Zuocheng Ren, Srinath Setty, Andrew J Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 341–357. ACM, 2013.

[24] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from rams to delegatable succinct constraint satisfaction problems. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 401–414. ACM, 2013.

[25] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *CRYPTO*, 2013.

[26] Jans Carlsson. snarklib: a c++ template library for zero knowledge proofs. https://github.com/jancarlsson/snarklib.

[27] bellman. `https://github.com/ebfull/bellman`.

[28] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. http://bitcoin.org/bitcoin.pdf, 2009.

[29] I. Bentov and R. Kumaresan. How to Use Bitcoin to Design Fair Protocols. In *CRYPTO*, 2014.

[30] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure Multiparty Computations on Bitcoin. In *S & P*, 2013.

[31] bitcoinj. `https://bitcoinj.github.io/`.

[32] R. Pass and a. shelat. Micropayments for peer-to-peer currencies. Manuscript.

[33] R. Kumaresan and I. Bentov. How to Use Bitcoin to Incentivize Correct Computations. In *CCS*, 2014.

[34] Augur. `http://www.augur.net/`.

[35] Skuchain. `http://www.skuchain.com/`.

[36] `http://koinify.com`.

[37] D. Mark, V. Zamfir, and E. G. Sirer. A call for a temporary moratorium on "The DAO" (v0.3.2). Referenced Aug. 2016 at http://bit.ly/2aWDhyY, 30 May 2016.

[38] K. Poulsen. Cybercrime supersite 'DarkMarket' was FBI sting, documents confirm. *Wired*, 13 Oct. 2008.

[39] A. Greenberg. Alleged silk road boss Ross Ulbricht now accused of six murders-for-hire, denied bail. *Forbes*, 21 November 2013.

[40] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *S&P*, 2015.

[41] Ben Kreuter, Benjamin Mood, Abhi Shelat, and Kevin Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Usenix Security Symposium*, pages 321–336, 2013.

[42] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2014.

[43] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S & P*, 2015.

[44] Joan Boyar, René Peralta, and Denis Pochuev. On the multiplicative complexity of boolean functions over the basis ($\wedge$, $\oplus$, 1). *Theoretical Computer Science*, 235(1):43–57, 2000.

[45] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.

[46] xJsnark website. `www.xjsnark.com`.

[47] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? `http://www.issi.uned.es/doctorado/generative/Bibliografia/Fowler.pdf`. Accessed: 11-01-2016.

[48] Xtext. `http://www.eclipse.org/Xtext/`.

[49] Jetbrains MPS. `https://www.jetbrains.com/mps/`.

[50] mbeddr. `http://mbeddr.com`.

[51] Youtrack. `https://www.jetbrains.com/youtrack/`.

[52] Markus Voelter. Language and IDE modularization and composition with MPS. In *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*, 2011.

[53] MetaR. `https://github.com/CampagneLaboratory/MetaR`.

[54] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering*, pages 41–61. Springer, 2014.

[55] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140. ACM, 2012.

[56] Vaclav Pech, Alex Shatalin, and Markus Voelter. Jetbrains mps as a tool for extending java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 165–168. ACM, 2013.

[57] Bruno Beauquier and E Darrot. On arbitrary size waksman networks and their vulnerability. *Parallel Processing Letters*, 12(03n04):287–296.

[58] Anup Hosangadi, Farzan Fallah, and Ryan Kastner. Optimizing polynomial expressions by algebraic factorization and common subexpression elimination. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):2012–2022, 2006.

[59] Robert K Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R Wang. Mis: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(6):1062–1081, 1987.

[60] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. Swifft: A modest proposal for fft hashing. In *International Workshop on Fast Software Encryption*, pages 54–72. Springer, 2008.

[61] Buffet's Merge Sort Benchmark. `https://github.com/pepper-project/pepper/blob/master/pepper/apps_sfdl/merge_sort.c`.

[62] LibZeroCash Github. `https://github.com/Zerocash/libzerocash`.

[63] Cédric Fournet, Chantal Keller, and Vincent Laporte. A certified compiler for verifiable computing. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*, pages 268–280. IEEE, 2016.

[64] Dorit Ron and Adi Shamir. Quantitative Analysis of the Full Bitcoin Transaction Graph. In *FC*, 2013.

[65] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of Bitcoins: characterizing payments among men with no names. In *IMC*, 2013.

[66] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *S & P*. IEEE, 2014.

[67] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *S & P*, 2013.

[68] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of finance*, 16(1):8–37, 1961.

[69] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *Advances in Cryptology–CRYPTO 2014*, pages 276–294. Springer, 2014.

[70] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Eurocrypt*, 2013.

[71] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, Hubert Chan, Charalampos Papamanthou, Rafael Pass, abhi shelat, and Elaine Shi. How to use snarks in universally composable protocols. `https://eprint.iacr.org/2015/1093.pdf`, 2015.

[72] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013.

[73] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey: An efficient mac algorithm for 32-bit microcontrollers. In *Selected Areas in Cryptography*. 2014.

[74] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *IEEE Symposium on Security and Privacy*, 2015.

[75] Mark Bagnoli and Barton L Lipman. Provision of public goods: Fully implementing the core through private contributions. *The Review of Economic Studies*, 56(4):583–601, 1989.

[76] Amazon EC2 pricing. `http://aws.amazon.com/ec2/pricing/`.

[77] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 305–326. Springer, 2016.

[78] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks. In *Annual International Cryptology Conference*, pages 581–612. Springer, 2017.

[79] Bitcoin ransomware now spreading via spam campaigns. `http://www.coindesk.com/bitcoin-ransomware-now-spreading-via-spam-campaigns/`.

[80] A. Greenberg. 'Dark Wallet' is about to make Bitcoin money laundering easier than ever. `http://www.wired.com/2014/04/dark-wallet/`.

[81] N. Christin. Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace. In *WWW*, 2013.

[82] A. Krellenstein, R. Dermody, and O. Slama. Counterparty announcement. https://bitcointalk.org/index.php?topic=395761.0, January 2014.

[83] http://www.smartcontract.com.

[84] Ethereum and evil. Forum post at Reddit; url: http://tinyurl.com/k8awj2j, Accessed May 2015.

[85] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. https://eprint.iacr.org/2015/460.

[86] https://github.com/darkwallet/darkleaks.

[87] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *FC*, 2014.

[88] NIST randomness beacon. https://beacon.nist.gov/home.

[89] Serpent. https://github.com/ethereum/wiki/wiki/Serpent.

[90] L. Bilge and T. Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In *CCS*, 2012.

[91] Electronic Frontier Foundation. EFF SSL observatory. URL: https://www.eff.org/observatory, August 2010.

[92] Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *S & P*, 2015.

[93] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1, 2003. RFC 3447.

[94] Verisign revoked certificate test page. https://test-sspev.verisign.com:2443/test-SPPEV-revoked-verisign.html. Accessed: 2015-05-15.

[95] CRL issued bby Symantec Class 3 EV SSL CA - G3. http://ss.symcb.com/sr.crl.

[96] E. F. Brickell, P. Gemmell, and D. W. Kravitz. Trustee-based tracing extensions to anonymous cash and the making of anonymous change. In *SODA*, volume 95, pages 457–466, 1995.

[97] M. Stadler, J.-M. Piveteau, and J. Camenisch. Fair blind signatures. In *Eurocrypt*, pages 209–219, 1995.

[98] Mt. Gox thinks it's the Fed. freezes acc based on "tainted" coins. (unlocked now). https://bitcointalk.org/index.php?topic=73385.0, 2012.

[99] J. Matonis. Why Bitcoin fungibility is essential. *CoinDesk*, 1 Dec. 2013.

[100] Blockchain Alliance. www.blockchainalliance.org, 2016.

[101] M. Peck. Ethereum developer explores the dark side of Bitcoin-inspired technology. *IEEE Spectrum*, 19 May 2016.

[102] RetractionWatch Website. `http://retractionwatch.com/`.

[103] Daniele Fanelli. How many scientists fabricate and falsify research? a systematic review and meta-analysis of survey data. *PloS one*, 4(5):e5738, 2009.

[104] Leslie K John, George Loewenstein, and Drazen Prelec. Measuring the prevalence of questionable research practices with incentives for truth telling. *Psychological science*, page 0956797611430953, 2012.

[105] The perfect Scientific Crime. `http://blogs.discovermagazine.com/neuroskeptic/2015/07/14/the-perfect-scientific-crime/#.WPjHa9Lytpk`.

[106] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2087–2104. ACM, 2017.

[107] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Bulletproofs: Short Proofs for Confidential Transactions and More*. IEEE Security and Privacy, 2018.

[108] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. A zero-knowledge version of vsql. Technical report, 2017.

[109] Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. IEEE Security and Privacy, 2018.

[110] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Technical report, 2018.

[111] Arjen K Lenstra. Using cyclotomic polynomials to construct efficient discrete logarithm cryptosystems over finite fields. In *Information Security and Privacy*, pages 126–138. Springer, 1997.

[112] Antoine Joux Razvan Barbulescu, Pierrick Gaudry and Emmanuel Thome. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In *EUROCRYPT'14*, 2014.

[113] Robert Granger, Thorsten Kleinjung, and Jens Zumbragel. On the discrete logarithm problem in finite fields of fixed characteristic. IACR ePrint Archive 2015/685, 2015.

[114] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. Cryptology ePrint Archive, Report 2005/133, 2005.

[115] Antoine Joux and Cécile Pierrot. The special number field sieve in F_{p^n}, application to pairing-friendly constructions. Cryptology ePrint Archive, Report 2013/582, 2013.

[116] Laurent Gremy Pierrick Gaudry and Marion Videau. Collecting relations for the number field sieve in $gf(p^6)$. IACR ePrint Archive 2016/124, 2016.

[117] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer, 2006.

[118] Daniel J. Bernstein and Tanja Lange. Safecurves: choosing safe curves for elliptic-curve cryptography. `http://safecurves.cr.yp.to`. Accessed: 2016-05-20.

[119] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.