xJsnark: A Framework for Efficient Verifiable Computation

Ahmed Kosba
UMD
akosba@cs.umd.edu

Charalampos Papamanthou
UMD
cpap@umd.edu

Elaine Shi
Cornell
rs2358@cornell.edu

Last Updated: October 13, 2019

***Abstract*—Many cloud and cryptocurrency applications rely on verifying the integrity of outsourced computations, in which a verifier can efficiently verify the correctness of a computation made by an untrusted prover. State-of-the-art protocols for verifiable computation require that the computation task be expressed as arithmetic circuits, and the number of multiplication gates in the circuit is the primary metric that determines performance. At the present, a programmer could rely on two approaches for expressing the computation task, either by composing the circuits directly through low-level development tools; or by expressing the computation in a high-level program and rely on compilers to perform the program-to-circuit transformation. The former approach is difficult to use but on the other hand allows an expert programmer to perform custom optimizations that minimize the resulting circuit. In comparison, the latter approach is much more friendly to non-specialist users, but existing compilers often emit suboptimal circuits.**

We present xJsnark, a programming framework for verifiable computation that aims to achieve the best of both worlds: offering programmability to non-specialist users, and meanwhile automating the task of circuit size minimization through a combination of techniques. Specifically, we present new circuit-friendly algorithms for frequent operations that achieve constant to asymptotic savings over existing ones; various globally aware optimizations for short- and long- integer arithmetic; as well as circuit minimization techniques that allow us to reduce redundant computation over multiple expressions. We illustrate the savings in different applications, and show the framework's applicability in developing large application circuits, such as ZeroCash, while minimizing the circuit size as in low-level implementations.

I. Introduction

Succinct Noninteractive ARguments of Knowledge (SNARK) [32], [43], [20], [14] is a powerful cryptographic building block that allows a prover to prove to a verifier the correctness of some computation, such that the verifier can check correctness in asymptotically less time than what it takes to perform the computation. SNARKs promise a broad spectrum of applications. For example, it allows a possibly computationally limited device to verifiably outsource computation to a powerful cloud server, problem that is often referred to as Verifiable Computation [43], [31], [46]. Known SNARK constructions also allow the prover to supply secret witnesses to the computation, thus enabling zero-knowledge proofs (henceforth referred to as zk-SNARKs). zk-SNARKs have been employed by various cryptocurrency systems such as ZeroCash [16], PinocchioCoin [26], and Hawk [36], to provide privacy-preserving transactions and/or smart contracts in a way that safeguards the integrity of the currency ledger's book-keeping and the correctness of smart contract execution.

SNARKs or (zk-SNARKs) support general computations, i.e., they can be used to prove the correctness of arbitrary, polynomial-sized computation. Most known SNARK constructions model computation as arithmetic circuits (or alternatively, as a set of arithmetic constraints) over a finite field. There are two existing methods for a programmer to express the computation that needs to be verified, either through compilation from a high-level language [43], [50], [25], or by manual circuit construction frameworks [20], [6]. 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 paper, 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 [50] and Geppetto [25]; and meanwhile improves the performance of the compiled SNARK implementation by **1.2**$\times$ to more than **3**$\times$ 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[16], while producing optimized output.

A. Problem Statement

An important goal of xJsnark is that of "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 [7], 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 [39], [38], [45], [41] — 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 [39], [38], [41], [45], [42], 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 [43], [17], [18], [25], [50], [26], [37] — but in this paper 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. 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.

B. 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 V) 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 [21]. 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 [17], [18], [50], 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 [27]); or adopt Karatsuba [35] that incurs $\Theta(m^{1.58})$ circuit size (e.g., ObliVM [39], GraphSC [42]) — 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, an 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 [27] that was built on top of Geppetto [25]. 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., 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.

C. 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 plan to open source our xJsnark framework in the near future.

Implementation. We present an overview of our xJsnark framework in Figure 1. xJsnark's front-end is developed atop Java using Jetbrains MPS, an open-source project [4] 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 [4]. 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 A, and discuss the trade-offs of using the underlying framework in Appendix D.

Using previous compilers like Buffet [50] or Geppetto[25], 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 VII-A). Additionally, we show how our framework produces efficient random access circuits by a factor of 2-3× in the case of sorting, while also providing more efficient ways to obtain more concise circuits (Section VII-B). Furthermore, we illustrate that the framework can produce efficient circuits as done by existing low-level implementations, as in the case of ZeroCash [16] when developed in our framework (Section VII-C).

II. Background and Related Work

In this section, we provide a necessary background to our paper, in which we cover the basics and related works on verifiable computation.

A (Zero-Knowledge) Succinct Noninteractive ARguments of Knowledge (SNARK) 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 $\vec{x}$ from $\mathcal{V}$, and (optionally) a secret input $\vec{u}$ from $\mathcal{P}$. $\mathcal{P}$ sends $\mathcal{V}$ both the output $\vec{y}$ and a proof $\vec{\pi}$ to verify the result. Specifically, a SNARK scheme typically consists of three algorithms [43]:

- $(PK_{\mathcal{F}}, VK_{\mathcal{F}}) \leftarrow \mathsf{KeyGen}(\mathcal{F}, 1^{\lambda})$: given an outsourced program $\mathcal{F}$ and a security parameter λ, output 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.

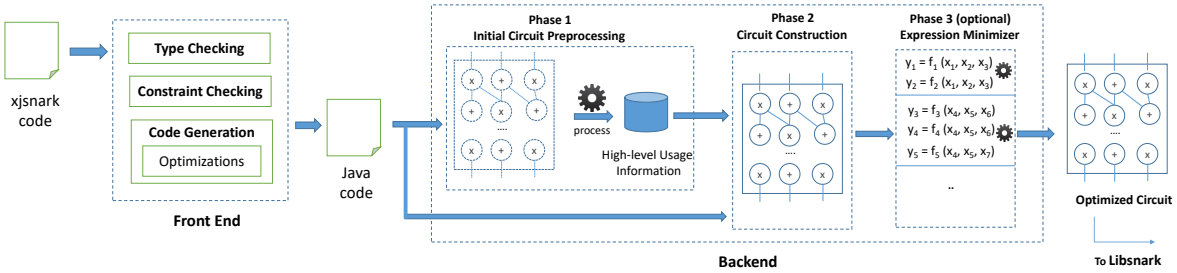

Fig. 1: xJsnark Overview

TABLE I: **Our contributions.** The part in blue highlights our new algorithms in support of each language feature.

	Bitwidth parametrized integers	Long integers	Memory access	Circuit minimization
Buffet [50]	Some	Library	Perm. net.	Some
Gepetto [25]	Some	Library	Does *not* support dynamic mem	Some
xJsnark	• Smart bitwidth alignment	• Improved algorithms for multiplication, subtraction, and equality	• New algorithm for read-only mem • Smart selection btw linear/read-only/perm net	• Multivar. polynomial minimization

- $(\vec{y}, \vec{\pi}) \leftarrow \mathsf{Prove}(\mathcal{F}, PK_{\mathcal{F}}, \vec{x}, \vec{u})$: given a program $\mathcal{F}$, the public proving key $PK_{\mathcal{F}}$, the public input $\vec{x}$, and the prover's secret input $\vec{u}$, output $\vec{y} \leftarrow \mathcal{F}(\vec{x}, \vec{u})$, and the proof $\vec{\pi}$ proving the correctness of the computation.
- $\{0,1\} \leftarrow \mathsf{Verify}(VK_{\mathcal{F}}, \vec{x}, \vec{y}, \vec{\pi})$: given the verification key $VK_{\mathcal{F}}$, the proof $\vec{\pi}$, and the statement $(\vec{x}, \vec{y})$, output 1 iff $\vec{y} = \mathcal{F}(\vec{x}, \vec{u})$.

In particular, for the scheme to satisfy succinctness, both the proof size and the time it takes to execute Verify must be asymptotically smaller that the time it takes to evaluate $\mathcal{F}$. 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 quadratic constraints, and a one-time setup per each different circuit is needed in the beginning. However, the techniques we describe in this paper can also be extended in a straightforward way to systems that support recursive composition of zk-SNARKs [19], [25]. In the interest of space, we omit the formal security definitions for (zk-)SNARKs, we refer interested readers to existing papers [32] for formal definitions.

A. 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) [32], [43], 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 expensive. 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 (some implementations add one constraint per a pack operation). For any two bits b_i, and b_j, the operations: AND, OR, and XOR each costs one constraint, 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 $p' = 2^n$, 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 the paper, the compiler can apply some heuristics for efficient translations of such operations. Previous compilers such as [43], [25], [50] assume that the programmer is responsible for taking such decisions.

Assertions and Comparisons (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 $a \times b = c$. This differs from checking the equality of two n-bit wires, which requires two constraints, while the unsigned integer $<,>$ comparison costs about $n+2$ constraints (assuming $n < \lceil \log_2 p \rceil - 1$).

Memory accesses. A program is different from a circuit in that it has control flows and dynamic memory accesses whose addresses cannot be determined at compile time. We will discuss how existing works and xJsnark translate memory accesses to circuits later in the paper Section V.

B. Existing Tools

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

High-level language compilers. This includes many works such as Pinocchio [43], TinyRam [18], Pantry [22], ZØ[30], Buffet [50], and Geppetto [25]. 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. The TinyRam compiler compiles high-level C programs to TinyRam assembly instructions (a simple RISC architecture), and generates arithmetic circuits that verifies the execution of TinyRam programs. In a later work[20], vnTinyRam generates a universal circuit that does not require a set up each time, but results in higher cost per program step. ZØ translates applications written in C# into code that produces scalable zero knowledge proofs of knowledge. It splits applications to distributed multi-tier code, and chooses between two zero-knowledge back-ends (Pinocchio and ZQL) to optimize performance.

Although the above tools include many theoretical and engineering optimizations, it is not straightforward to develop programs efficiently for zk-SNARKs, 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 [16], [36], [34]. This for example includes libsnark's gadget libraries [7], and jsnark [6]. 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 [24] , and bellman [1].

III. xJsnark's Front End

In this section, we discuss the language extension features, in addition to some optimizations that can be applied early during the translation to Java code process. 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 A[1]. Background and discussion of Jetbrains MPS are provided in Appendix D.

A. Extension Features

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 3v;
F_swifft y1 = 123;
F_p256 y2 = 81052781576157641653615236513 1v;
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.

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
  }
}
```

[1]The latest versions of the examples using up-to-date syntax specifications can be found in the xJsnark website [11].

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) External Code Blocks: In many cases, computing the value of prover's witnesses 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 A-B).

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 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 A-B for an example, and Section VII-B for performance results).

We provide additional technical details on type and syntactic checking, control flow, code generation and additional extension features in the appendix (Appendix F).

IV. 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.

A. 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.

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 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 (in order): Ensuring no overflows can happen in later operations involving e, and minimizing the total cost resulting from bitwidth adjustments.

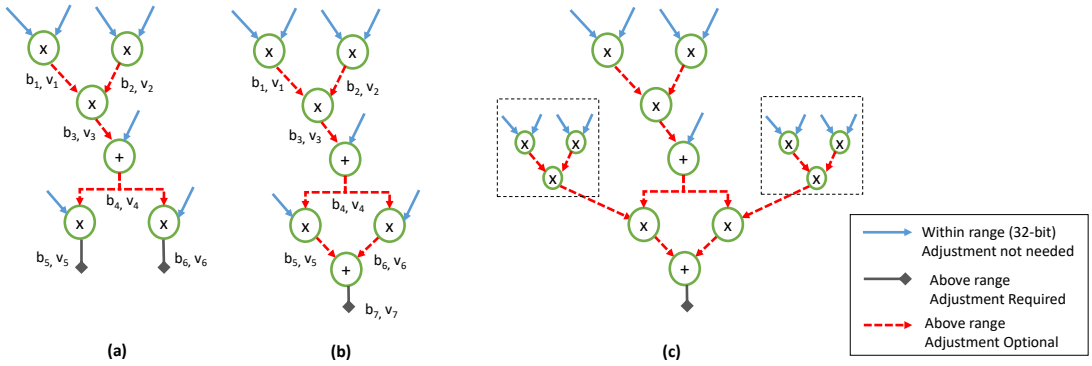

Fig. 2: Bitwidth adjustment examples

This can be modeled as a constrained optimization problem. To illustrate that by example, Figure 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 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

$$
\begin{aligned}
v_1 =& v_2 = 64 \\
b_5 =& b_6 = 1 \\
v_3 =& v_1 + b_1(32 - v_1) + v_2 + b_2(32 - v_2) \\
v_4 =& v_3 + b_3(32 - v_3) + 1 \\
v_5 =& v_6 = v_4 + b_4(32 - v_4) + 32 \\
v_i \in& \{32, .., \lfloor \log_2 p \rfloor - 1\} \\
b_i \in& \{0, 1\}
\end{aligned}
$$

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 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 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 many 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.

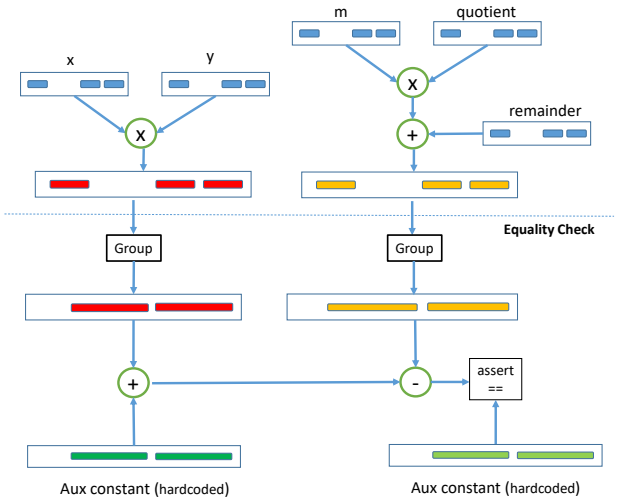

Fig. 3: Equality assertion in a modular multiplication circuit

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.

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 mod 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) 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.

B. 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 $b < \log_2 p$.

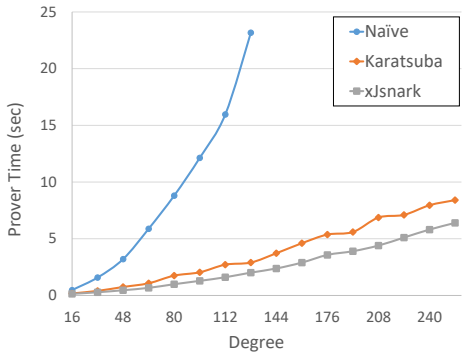

Fig. 4: Long integer multiplication methods

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.

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 [35], 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 $c \in \{1, .., 2m-1\}$, 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 $2m-1$ independent and consistent 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 4 illustrates a comparison between the three approaches above with respect to the proof time on a single processor.

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) 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.

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 [25], [37].

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. 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 [25], [37]. 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 [25], [37] 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

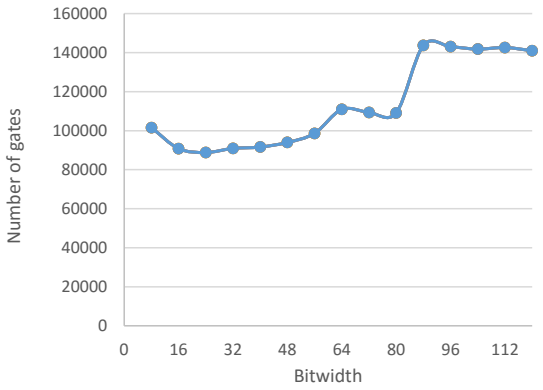

Fig. 5: Bitwidth Effect on Number of Gates in RSA modular exponentiation circuit

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 [27]), 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 5).

V. 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 [43], [50]:

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 [15]: This approach costs $O((k + n) \log(k + n))$ for k accesses where the starting memory size is n. This approach was proposed in [17], and subsequently used in TinyRAM [18], and Buffet [50].

In Appendix B-A, we discuss low-level optimizations for both Merkle tree and permutation network approaches.

A. Algorithm for read-only memory access

In this section, we discuss a new method for implementing dynamic memory accesses in read-only arrays whose contents

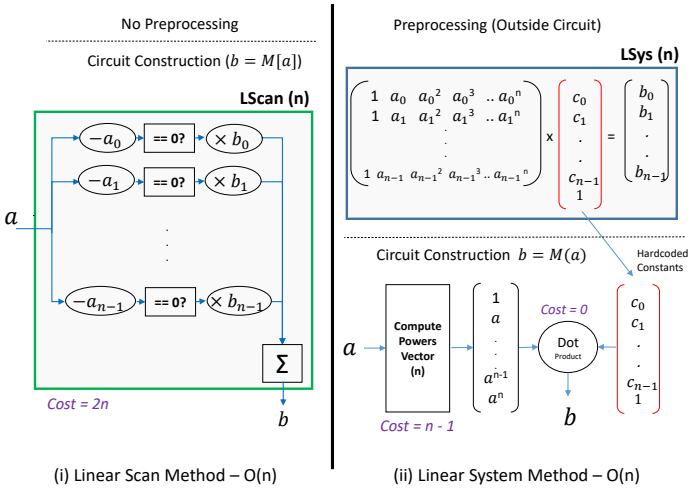

Fig. 6: $O(n)$ methods for read-only memory access

are prepopulated. 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 and trigonometric 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, however, 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 — in fact, we can make further optimizations that improve the performance by $O(1)$ factor by making use of the fact that the array contents are known at compile time (i.e., constants), and multiplication with a constant comes for free in our cost model. When comparing with these existing techniques, Table II will assume that these optimizations have been applied to existing techniques.

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

Building block: polynomial function via a linear system solution. We first describe a new 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.

Intuition. The proposed technique relies on the power of SNARK verification. 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 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 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

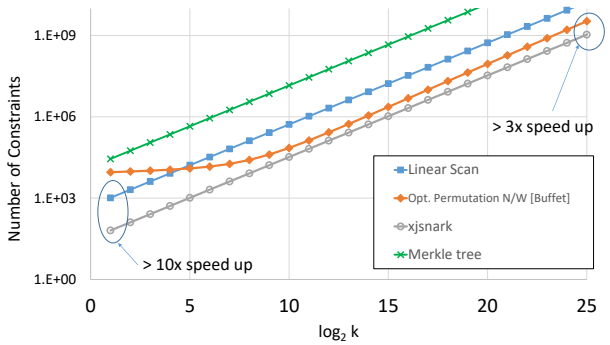

Fig. 7: Comparison between the proposed $O(\sqrt{n})$ method for read-only memory access, and other optimized existing approaches when $n = 256$. k represents the number of accesses.

$z = b + a.n$ 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 8 in Appendix B-B. The actual cost in the circuit will be equal to $2\sqrt{n} + \log_2 n$ constraints per access. Further optimization that reduces the cost by a factor of 2 for applications like AES is provided in Appendix B-B.

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) The total cost of applying the permutation network is $O((n+k)(\log(n+k)))$ as mentioned earlier, which means that the cost of accessing an element also depends on the number of accesses made in addition to the memory size. Table II compares all the techniques discussed so far.

Case study when n = 256. Figure 7 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 $10\times$ when the number of accesses is 2, to more than $3\times$, when the number of memory accesses is more than 32 million.

[2]Note that xJsnark also switches to the permutation network approach when it is more efficient, based on the workload analysis in the first phase. See Section V-B for details.

TABLE II: 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	$O(k \log n)$	$2000k \log_2 n$
Perm. N/w ([50])	$O((n+k)(\log(n+k)))$	$(n+k)(\log_2(n+k))+$ $2\log_2(k+n)+3\log_2 n$
xJsnark[2]	$O(k\sqrt{n})$	$k(2\sqrt{n}+\log_2 n)$

B. Smart Memory Implementation

In our framework, a special syntax is used to instantiate a random access memory, however the programmer will still be able to use the typical array operators for accessing the memory. In the first preprocessing stage of the back end, each memory is studied separately, and the compiler takes the following factors into account: 1) The number of read/write operations, 2) the type/size of data being accessed, and 3) whether the memory contents are 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.

VI. 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: $maj_i = t_i + c_i(a_i + b_i - 2t_i)$, where $t_i = a_i b_i$.

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 [33] 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. Due to space constraints, we provide the details in Appendix C.

VII. 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 discuss savings for random memory access. Furthermore, the evaluation also includes the full large circuit used by ZeroCash [16] 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.

A. 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, [25], [50]. 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 [16] and Hawk [36], mostly in a *manually* optimized way built using either libsnark [7] or jsnark [6], which provide a circuit that has approximately 27,100 and 26,000 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 III) indicate that the SHA-256 circuit produced automatically by xJsnark achieves (**1.5**$\times$ and **1.7**$\times$) 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 [40], 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 III, 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 III 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 [27] for signature verification, 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

TABLE III: 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. For AES, the number between parentheses indicates the number of blocks, and the cost includes the cost of key expansion.

	Buffet [50]		Geppetto [25]		xJsnark	
SHA-256	44999	○	38556	○	25538	○
SWIFFT	3857	◐	3006†	●	3006	○
RSA-2048	-	●	144933 [27]	◐	88949	○
AES-128 (1)	50000†	○	90000†	◐	14240	○
AES-128 (300)	10×10^6†	○	27.2×10^6†	◐	3.5×10^6	○

witness inputs to the circuit.

To ensure a fair comparison, we implemented the specification provided in the Cinderella paper (assuming that the modulus is hardcoded in the circuit, and the exponent is set to 65537), and compared it with our back end technique described earlier in the same setting. Note that the cost of our Cinderella baseline implementation is less than the cost of the original implementation [27]. In Cinderella's specification, big integers are divided into 120-bit words, and hence the grouping step described in our equality assertion algorithm cannot be applied, 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 grouping step. The result is shown in Table III, showing more than **1.5**× speed-up **overall**. Looking closer, the enhancement in the 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 III illustrates the results, when the number of AES blocks is 1 and 300. 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 [50], which uses an unoptimized permutation network. As shown in the table, our memory approach provides more than **2** to **3**× speedup over Buffet for the whole AES circuit. It should also be noted that our $O(\sqrt{n})$ approach used in this evaluation achieves **1.7**× enhancement over Buffet with a more optimized permutation network implementation. The savings also apply to the key sizes and the memory usage.

TABLE IV: Number of constraints for sorting circuits (n: input size)

n	Buffet[50] 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[50]	3.8×10^6	14830

TABLE V: 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[16], [8]	●
xJsnark	3.81×10^6	○
Buffet [50]	6×10^6†	○
Geppetto [25]	5×10^6†	○

B. 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 [50], and xJsnark. The first two columns of Table IVcompare the circuit sizes produced by a merge sort implementation of an array of 16-bit integers, that is according to Buffet's available repository [2]. 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**× 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 III. This method provides significantly better results as it saves a logarithmic factor of comparisons, and eliminates the cost of read/write memory operations. Table IV shows the savings compared to basic approaches (Appendix A-C provides a code example).

C. ZeroCash's ZK-SNARK Circuit

Using our framework, we developed one existing application that was manually developed using the libsnark gadget library [7], [8], mainly the pour circuit in the ZeroCash system [16], which is used to add privacy to transactions on top of the blockchain.

Table V 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 VI). In terms of the development effort, the implementation is more compact in comparison with the existing available implementation online on Github [8], and the gadgets it uses from libsnark [7]. The rest of the table shows the efficiency achieved by xJsnark compared to other compilers.

A detailed discussion of limitations and future work is provided in Appendix E. The full version of the paper and additional code examples will be made available on this website [11].

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their suggestions. This work was supported in part by NSF awards #1514261, #1652259 and #1617676, and by a NIST award.

REFERENCES

[1] bellman. https://github.com/ebfull/bellman.
[2] Buffet's Merge Sort Benchmark. https://github.com/pepper-project/pepper/blob/master/pepper/apps_sfdl/merge_sort.c.
[3] Jetbrains MPS. https://www.jetbrains.com/mps/.
[4] Jetbrains MPS Github. https://github.com/JetBrains/MPS.
[5] jsnark - AES Implementation. https://github.com/akosba/jsnark/tree/master/JsnarkCircuitBuilder/src/examples/gadgets/blockciphers.
[6] jsnark: A java library for building snarks. oblivm.com/jsnark.
[7] libsnark. https://github.com/scipr-lab/libsnark.
[8] LibZeroCash Github. https://github.com/Zerocash/libzerocash.
[9] mbeddr. http://mbeddr.com.
[10] MetaR. https://github.com/CampagneLaboratory/MetaR.
[11] xJsnark website. www.xjsnark.com.
[12] Xtext. http://www.eclipse.org/Xtext/.
[13] Youtrack. https://www.jetbrains.com/youtrack/.
[14] M. Backes, M. Barbosa, D. Fiore, and R. M. Reischuk. Adsnark: nearly practical and privacy-preserving proofs on authenticated data. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015.
[15] B. Beauquier and E. Darrot. On arbitrary size waksman networks and their vulnerability. *Parallel Processing Letters*, 12(03n04):287–296.
[16] 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*, 2014.
[17] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. 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.
[18] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *CRYPTO*, 2013.
[19] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, 2014.
[20] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security*, 2014.
[21] J. Boyar, R. Peralta, and D. Pochuev. On the multiplicative complexity of boolean functions over the basis ($\wedge$, $\oplus$, 1). *Theoretical Computer Science*, 235(1):43–57, 2000.
[22] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 341–357. ACM, 2013.
[23] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. 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.
[24] J. Carlsson. snarklib: a c++ template library for zero knowledge proofs. https://github.com/jancarlsson/snarklib.
[25] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *S&P*, 2014.
[26] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *PETShop*, 2013.
[27] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno. Cinderella: Turning shabby x.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *S& P*, 2016.
[28] C. Fournet, C. Keller, and V. Laporte. A certified compiler for verifiable computing. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*, pages 268–280. IEEE, 2016.
[29] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? http://www.issi.uned.es/doctorado/generative/Bibliografia/Fowler.pdf. Accessed: 11-01-2016.
[30] M. Fredrikson and B. Livshits. Zø: An optimizing distributing zero-knowledge compiler. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 909–924, 2014.
[31] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, CRYPTO'10, pages 465–482, 2010.

[32] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct nizks without pcps. In *Advances in Cryptology–EUROCRYPT 2013*, pages 626–645. Springer, 2013.
[33] A. Hosangadi, F. Fallah, and R. 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.
[34] A. Juels, A. Kosba, and E. Shi. The ring of gyges: Using smart contracts for crime. Manuscript, 2015.
[35] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.
[36] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy*, 2016.
[37] A. Kosba, Z. Zhao, A. Miller, Y. Qian, H. Chan, C. Papamanthou, R. Pass, abhi shelat, and E. Shi. C∅c∅: A framework for building composable zero-knowledge proofs. Cryptology ePrint Archive, Report 2015/1093, 2015. http://eprint.iacr.org/2015/1093.
[38] B. Kreuter, B. Mood, A. Shelat, and K. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Usenix Security*, 2013.
[39] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *S&P*, 2015.
[40] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. Swifft: A modest proposal for fft hashing. In *International Workshop on Fast Software Encryption*, pages 54–72. Springer, 2008.
[41] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay: a secure two-party computation system. In *USENIX Security*, 2004.
[42] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S & P*, 2015.
[43] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *S & P*, 2013.
[44] V. Pech, A. Shatalin, and M. 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.
[45] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *S&P*, 2014.
[46] E. Shi, A. Perrig, and L. V. Doorn. BIND: A fine-grained attestation service for secure distributed systems. In *IEEE Symposium on Security and Privacy*, 2005.
[47] M. 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.
[48] M. Voelter, D. Ratiu, B. Schaetz, and B. 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.
[49] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering*, pages 41–61. Springer, 2014.
[50] R. S. Wahby, S. T. V. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, 2015.

APPENDIX A
ADDITIONAL CODE EXAMPLES

In this section, example code written using our framework is provided. Additional examples using up-to-date syntax specifications can be found in the xJsnark website [11].

A. Simple Circuit Example

The following piece of code specifies a simple circuit that performs a dot product operation:

```
Program SimpleCircuit {
  // SIZE denotes the vector size
  uint_32[] x = new uint_32[SIZE];
  uint_32[] y = new uint_32[SIZE];
  uint_32 z;

  inputs {x, y}; // circuit inputs
  outputs {z};   // circuit outputs
  witnesses {};  // external prover input

  void Main(){
    z = 0;
    for(int i = 0; i < SIZE; i++){
      z = z + x[i]*y[i];
    }
  }
}
```

B. External Code Example

The following piece of code specifies a circuit, where a solution is desired for a simple 2×2 linear system of equations:

```
Program LinearSystemVerification {
  uint_32[] a1 = new uint_32[2];
  uint_32[] a2 = new uint_32[2];
  uint_32 b1;
  uint_32 b2;
  uint_32[] solution = new uint_32[2];

  inputs{ a1,a2,b1,b2 };
  outputs { };
  witnesses {solution};

  void Main(){
    external {
      // read the values of the variables during runtime, and
      // convert them to the BigInteger Java type
      BigInteger[] a1_vals = new BigInteger[]{a1[0].$val$,
          a1[1].$val$};
      BigInteger[] a2_vals = new BigInteger[]{a2[0].$val$,
          a2[1].$val$};
      BigInteger[] b_vals = new BigInteger[]{b1.$val$, b2.$val$};

      BigInteger[] solution_vals = solve(a1_vals, a2_vals, b_vals);
      solution[0].$val$ = solution_vals[0];
      solution[1].$val$ = solution_vals[1];
    }
    verifyEq(solution[0]*a1[0] + solution[1]*a1[1], b1)
    verifyEq(solution[0]*a2[0] + solution[1]*a2[1], b2)
  }

  BigInteger[] solve( .. ){
    // A Java method that solves linear systems of equations over
        finite fields
  }
}
```

C. Sorting Code Example

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.

```
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] );
  }
}
```

APPENDIX B
ADDITIONAL DETAILS FOR MEMORY IMPLEMENTATION

A. Optimizations for earlier methods

In this section, we discuss optimizations for both the Merkle tree and the permutation network approaches:

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 [19], and later analyzed in [37]. Using such hash function, the cost per level can be 2032 gates to achieve more than 128 bit security level.

2) Permutation network approach: A permutation network is typically implemented as an AS-Waksman network 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.

In some situations involving small memories and short data elements, it might 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. 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 $(w_1 - w'_1)(w_2 - w'_1) = 0$, 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.

B. The Read-only memory case

Additional Illustration Figure 8 provides an additional visual illustration of the read-only memory approach in Section V-A.
Additional Optimizations. Further optimizations can be made to the earlier approach to reduce the number of gates per access, 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. This has to ensure that there is only a unique valid answer per each look-up. 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 the AES implementation, since the bit decomposition of a and b are typically needed for other parts in the circuit. In our implementation (a low-level version of the implementation is available in jsnark [5]), a single look-up will only require adding 16 additional constraints.

APPENDIX C
ARITHMETIC OPTIMIZATION MODULE DETAILS

In this section, we illustrate the details of the arithmetic optimization module.

A. 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 the output cannot be written as a polynomial function 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.

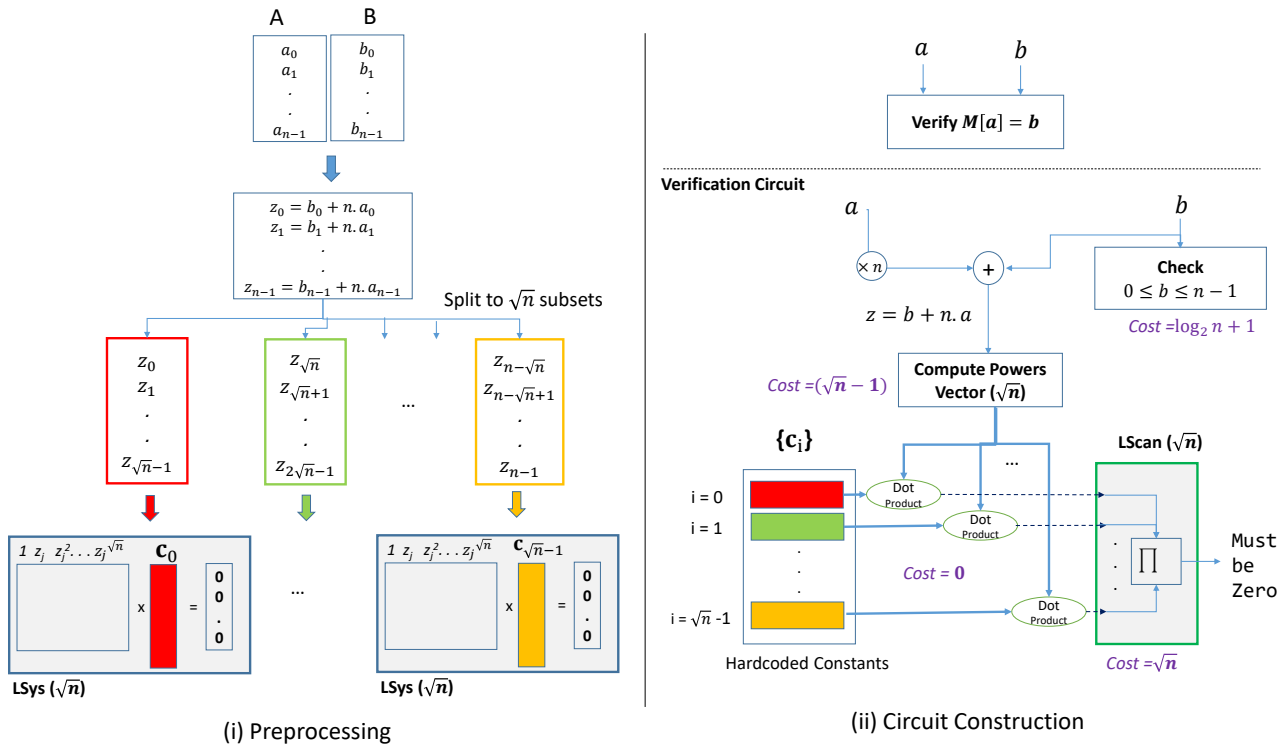

Fig. 8: Additional illustration of the read-only memory approach in Section V-A

- 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. One approach would 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 9 illustrates an example of how the wires of a simple circuit are labeled.

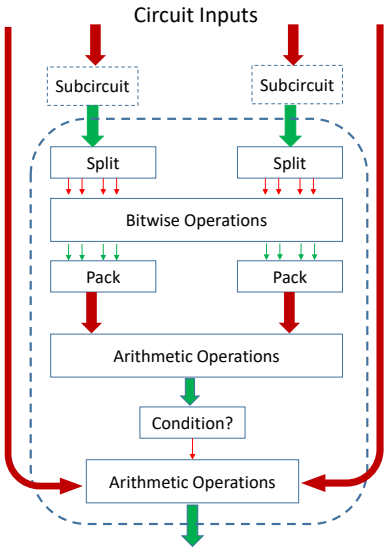

Fig. 9: How opt-input and opt-output wires are selected. Red wires indicate opt-inputs, and green wires indicate opt-outputs.

B. *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.

We would ideally like to perform optimizations over all expressions extracted from the circuit. However, since the multivariate polynomial minimization algorithm we rely on runs in worst-case exponential time, in practice this global approach would be too expensive. 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, 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.

C. 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 [33], which is already based on known techniques in Multi-level logic synthesis, such as [23]. 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 [33] is that we distinguish between multiplication of variables, and multiplication by constants, to suit the cost model described earlier in the background section. Additionally, we added a brute force exploration module within the algorithm that can help with small problem sizes. Finally, we add a tunable parameter for the programmer to control the level of exploration required for the optimization.

D. 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 [33]. Another limitation is its running time and memory consumption for large problems, therefore, we restrict the size of the problems tackled by this module.

Appendix D
Additional Background: Language extension and Jetbrains MPS

Our language extension for the front end is built using Jetbrains MPS [4], an open source language workbench [29] 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 [12], and projectional editor-based workbenches such as Jetbrains MPS that we use here [3]. 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* [9] which provides extensions on top of the C language for embedded system development. Other examples include Youtrack [13], an issue tracking system that has a Java language extension for working with persistent data and queries among others [47], and MetaR which facilitates biological data analysis with the R language [10]. 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 [49], while the mbeddr authors [48] 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 (Appendix E), 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 [44]. To define a language extension, such as the one that we use in this paper, 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 AST 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 equations needed for any introduced new types.
Code generation: This specifies how the extensions 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.

APPENDIX E
DISCUSSION 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 discussed in this paper. 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 [28].

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.

3. Dynamic Pointer/Reference Assignments [3]**.** Our implementation does not allow manipulating references to xJsnark's structs (which correspond to pointers in C) within code blocks that rely on a circuit run-time conditional check. Although the Buffet compiler [50] supports it, it might not be implemented in the most optimized way, as it does not take into account how the pointers are being used. Additionally, in some cases, it might be more efficient to cluster pointer accesses into groups, and study each separately. One direction of future work will be how to analyze pointer usage in the first pass automatically, and integrate our adaptive memory back end algorithms with pointer manipulation.

[3]This limitation has been resolved in our most recent version, through the runtime struct feature, which will be explained in detail with examples in a future version

APPENDIX F
ADDITIONAL FRONT-END DETAILS

A. Extension Features (continued)

1) Composite Structures: This allows the programmer to define a collection of xJsnark's introduced native data types or other composite structures. Such structures have a special implementation in the back end, so that the programmer can easily manipulate them as circuit inputs/outputs. The programmer uses the keyword `struct` to define such special classes. This will be illustrated in the zerocash coding example.

2) 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 `verifyNotEqual`, `verifyZero` and `verifyNonZero`.

B. Type and Syntactic Constraint Checking

Introducing new types and features requires additional type rules and constraints to be enforced/checked by the front end. Type checking rules includes checking bitwidth properties during assignments and operations. For simplicity, the following code snippet shows few examples illustrating the type checks:

```
..
uint_19 a;
uint_33 b;
F_swifft f1;
F_p256 f2;
..
a = b; // error
b = a; // allowed
uint_33 c = a + b; // type of (a+b) is uint_33
f1 = f2; // error
f1  = f1+f2; // error
```

Examples for constraint checking include ensuring the `external` code block appearing within a method definition, and that `$val$` operator is only used when inside that block. Another example is to only use variables declared as xJsnark types inside the input, output and witness blocks.

C. 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, one naive approach would be to compute the result of the conditional statement as a bit variable, multiply it by other bits from outer conditional statements (if any), and then use this bit variable in all operations, such that 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 [41] compiler. This can do potential savings in nested if statements, and when multiple operations are inside a block. Since we are not building the compiler from the ground-up, we implemented the algorithm differently, by adding instructions in both in the code generation, and adding classes in the back end that apply the algorithm, only on xJsnark types. It has the same complexity as in the fairplay compiler [41].

APPENDIX G
ZERO-CASH'S POUR CIRCUIT

As a case study, we illustrate how the framework can be used to program the pour circuit of ZeroCash [16], 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 [16], 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 [8], 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[16] for better readability (The reader may consult Figure 2 in the ZeroCash Paper for the detailed specifications).

A. 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;
  }
}
```

B. 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. 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 [8] and its gadget dependencies in [7], it can be observed how xJsnark saves a lot of the programming effort, while producing efficient output as illustrated in Section VII-C.