Succinct Non-Interactive Arguments for a von Neumann Architecture

Eli Ben-Sasson
Technion

Alessandro Chiesa
MIT

Eran Tromer
Tel Aviv University

Madars Virza
MIT

December 30, 2013

Abstract

We design and build a system that enables clients to verify the outputs of programs executed by untrusted servers. A server provides a succinct non-interactive zero-knowledge proof (also known as a *zk-SNARK*), which the client verifies to ascertain correct execution.

The system has two components: a cryptographic proof system for verifying satisfiability of arithmetic circuits, and a circuit generator to translate program executions to such circuits. Our design of both components improves in functionality and efficiency over previous work, as follows.

Our circuit generator is the first to be *universal*: it does not need to know the program, but only a bound on its running time. It is also the first to support programs expressed as code for a *von Neumann RISC* random-access memory architecture, where programs may use just-in-time compilation and self-modifying code. Moreover, the dependence on program size is additive (instead of multiplicative as in prior works), allowing efficient verification of *large programs*.

The cryptographic proof system significantly improves proving and verification times, using a new pairing-based cryptographic library tailored to the protocol.

We evaluated our system for programs with up to 10,000 instructions, running for up to 32,000 machine steps, each of which can arbitrarily access random-access memory; and demonstrated it executing programs that use just-in-time compilation. Our proofs are 230 bytes long at 80 bits of security, or 288 bytes long at 128 bits of security. Typical verification time is 5 ms, regardless of the original program's running time.

Keywords: computationally-sound proofs, succinct arguments, zero-knowledge, delegation of computation

Contents

1 Introduction

Consider the setting where a client owns a public input x and a server owns a private input w. The client wishes to outsource the computation of a publicly-known program F to the server, in order to learn $z := F(x, w)$, the correct output of F on the two inputs (x, w). (A special case of this setting is when w is the empty string: the client outsources the computation of $z := F(x)$.)

Two main security desiderata arise in this setting:

(1) INTEGRITY OF COMPUTATION. How can the client ascertain that the server reported the correct output?

(2) CONFIDENTIALITY OF THE SERVER'S INPUT. How can the server prevent the client from learning anything about his input w (beyond what is inevitably revealed by z)?

Moreover, the client may be either unable or unwilling to engage in lengthy interactions with the server, or to perform large computations beyond the "bare minimum" of sending the input x and receiving the output z. For instance, the client may be a computationally-weak device with intermittent connectivity, such as a smartphone. Hence, we consider the following two additional *efficiency* desiderata.

(3) NON-INTERACTIVITY. It suffices for the server to send the client the claimed output $\tilde{z}$, along with a non-interactive proof π that attests that $\tilde{z}$ is the correct output.

(4) SUCCINCTNESS. The client runs in time that is proportional only to (i) the size of F, (ii) the length of his input x, and (iii) the length of the output z. But the client's running time does *not* depend on the length of the server's input w, nor the running time of F on input (x, w).

The scenario above is very general, and cryptographic protocols whose guarantees address the aforementioned security and efficiency desiderata can be applied to a wide range of security applications — provided these protocols deliver good enough *efficiency*, and support rich enough *functionality* (i.e., the class of programs F that is supported).

1.1 Goal

To achieve the desiderata, we seek a *proof system*, between a *prover* and a *verifier*, which works as follows.

- A proof is *non-interactive*: it consists of a single message π from the prover to the verifier. The server (acting as the prover) can then convince the client (acting as the verifier) that he knows w for which $z = F(x, w)$ by generating a proof π for the statement "there exists w s.t. $z = F(x, w)$". By verifying π, the client checks that the server did not lie.
- A proof is *zero knowledge*: π leaks no information about w, thereby ensuring confidentiality of the server's input.
- A proof is a *proof of knowledge*. Intuitively, this means that if the client receives a valid proof π from the server, he can deduce not only that the statement is true, but also that the server *knows* a valid w that explains the output z.
- A proof is *succinct*, i.e., it is very short and easy to verify.

A proof system that achieves the above properties is called a (publicly-verifiable) *zero-knowledge Succinct Non-interactive ARgument of Knowledge* (zk-SNARK) (see Section 3.2.)

Our goal in this paper is a zk-SNARK implementation, and we strive for good *efficiency* and rich *functionality* of supported programs.

1.2 Prior work

There are numerous works studying variations or relaxations of the goal considered in this paper; in Section 7 we summarize some of these. Many works have obtained zk-SNARK constructions [Gro10a, Lip12, GGPR13, BCI$^+$13, PGHR13, BCG$^+$13, Lip13]. Two of these [PGHR13, BCG$^+$13] also provide implementations; these two works are the most relevant to us, and thus we briefly recall them.

Parno et al. [PGHR13] present two main contributions.

- A zk-SNARK, with essentially optimal asymptotics, for arithmetic circuit satisfiability. They accompany their construction with an implementation. Their construction is based on *quadratic arithmetic programs* (QAPs) [GGPR13].
- A compiler that maps C programs with fixed memory accesses and bounded control flow (e.g., array accesses and loop iteration bounds are compile-time constants) into corresponding arithmetic circuits.

Ben-Sasson et al. [BCG$^+$13] present three main contributions.

- Also a zk-SNARK with essentially optimal asymptotics for arithmetic circuit satisfiability, and with a corresponding implementation. Their construction is also based on QAPs, but follows the linear interactive proof approach of [BCI$^+$13].
- A simple RISC architecture, TinyRAM, along with a circuit generator for generating arithmetic circuits that verify correct execution of TinyRAM programs.
- A compiler that, given a C program, produces a corresponding TinyRAM program.

Thus, both works have a component consisting of a zk-SNARK for a low-level language, and a component for translating a high-level language to the low-level language.

1.3 Limitations of prior work

Prior work has made tremendous progress in improving the efficiency and functionality of zk-SNARK implementations. Yet, known implementations suffer from two main limitations.

Support for high-level languages remains limited. The circuit generator in [PGHR13] only supports programs without data dependencies: it does not allow memory accesses and loop iteration bounds to depend on a program's input. This limitation is significant, since data dependencies are common.

The circuit generator in [BCG$^+$13] does support arbitrary programs, but it too suffers from significant limitations. First, programs need to be compiled to TinyRAM, which, being a word-addressable Harvard architecture, is inconvenient to work with. But, more importantly, the circuit generator outputs circuits with $\Omega(\ell T)$ gates for ℓ-instruction T-step TinyRAM programs. Clearly, a size of $\Omega(\ell T)$ cannot scale.

Indeed, as zk-SNARK implementations get faster and more scalable, users may wish to verify execution of more sophisticated programs. Inevitably, such programs will rely on libraries (providing, e.g., mathematical subroutines or data structures), which contribute to program size. Thus, it is crucial for circuit generators to produce circuits that efficiently scale with program size.

Prior implementations are "1st generation". zk-SNARKs are *pairing-based protocols*. We know that protocol-dependent optimizations are a key ingredient for fast implementations of such protocols [Sco05]. Prior implementations only utilize off-the-shelf cryptographic libraries, and thus miss key optimization opportunities, especially in the verifier.

High-performance implementations of zk-SNARKs, even for a low-level language such as arithmetic circuit satisfiability, are a central goal: such implementations provide strong foundations both for systems that prove/verify "circuit-like" computations, as well as for systems that extend functionality to complex programs via circuit generators.

1.4 Results

In this paper we present two main contributions, each addressing a limitation of prior work.

(1) We design and build a **new circuit generator** that incorporates the following three main improvements.

Improvement #1: Our circuit generator supports programs on a *new, more expressive architecture*: vnTinyRAM. Unlike TinyRAM, our architecture follows the *von Neumann paradigm*: program and data are stored in the same read-write address space, as in modern machines. Programs may thus utilize techniques such as *just-in-time compilation* or *self-modifying code*. Another change is that memory is accessible as either bytes or words, to efficiently support both string operations and integer arithmetic.

vnTinyRAM can thus efficiently support many programming styles, including (an adaptation of) the C compiler of [BCG$^+$13]. We take program executions on vnTinyRAM to be the "high-level language" in this paper.

Improvement #2: Our circuit generator is *universal*: when given input bounds ℓ, n, T, it produces a circuit that can verify the execution of *any* program with $\leq \ell$ instructions, on *any* input of size $\leq n$, for $\leq T$ steps. In contrast, prior circuit generators [PGHR13, BCG$^+$13] hardcoded the program in the circuit.

Improvement #3: Our circuit generator *efficiently handles large programs*. Concretely, the size of the generated circuit in terms of the bounds ℓ, n, T, is

$$O\big((\ell + n + T) \cdot \log(\ell + n + T)\big) \text{ gates.}$$

When $\ell = \Omega(T)$, this gives a *quadratic improvement* over [BCG$^+$13], where the generated circuit has size $\Theta\big((n + T) \cdot (\log(n + T) + \ell)\big)$.

Our efficiency improvements are not merely asymptotic but yield concrete savings. For example, the amortized number of gates per step is nearly unaffected by program size:[1]

- $\approx$ 1,420 gates/step for $(\ell, n, T) = (10^4$,100,$2^{20})$, vs.
- $\approx$ 1,442 gates/step for $(\ell, n, T) = (10^5$,100,$2^{20})$.

These numbers improve $7.6\times$ and $70.0\times$ respectively over the per-cycle counts of the circuit generator in [BCG$^+$13]; see Figure 1. Moreover, our superior asymptotics ensure that the efficiency improvement continues to grow, without bound, as the program size bound ℓ increases. (In fact, even for a million-instruction program running for a billion steps, we pay only $\approx 1{,}399$ gates per cycle!)

$n = 10^2, T = 2^{20}$	[BCG$^+$13]	**this work**	improvement
$\ell = 10^3$	1,872	1,418	1.3$\times$
$\ell = 10^4$	10,872	1,420	7.6$\times$
$\ell = 10^5$	100,872	1,442	70.0$\times$
$\ell = 10^6$	1,000,872	1,661	602.6$\times$

Figure 1: Per-cycle count (i.e., $\frac{\text{\# gates}}{T}$) improvements over [BCG$^+$13].

[1]Note that T counts steps of the *abstract* machine vnTinyRAM. Each step has a *concrete* cost k in gates, even when merely executing the machine without any verification. Typically, in our circuit generator $> 50\%$ of the gates are dedicated to execution, and the remainder to verification. Thus, our "multiplicative overhead" compared to the concrete cost kT is only ≈ 2.

A comparison with the circuit generator in [PGHR13] is not well-defined: (i) it only supports programs with no data dependencies (so a programmer must “write around” the limited functionality via inefficient implementations of primitives such as array accesses); and (ii) its efficiency is not easily specified. Intuitively, we expect [PGHR13]’s circuit generator to perform better for programs that are “already closer to a circuit”, and worse for programs richer in functionality (which comprise most programs in practice). Experimentally, this is what we find; see Section 6.2.

(2) A **high-performance implementation of a zk-SNARK for arithmetic circuit satisfiability**. We improve upon and implement the protocol of Parno et al. [PGHR13]. Unlike previous zk-SNARK implementations [PGHR13, BCG$^+$13], we do not rely on any off-the-shelf cryptographic libraries. Instead, we build from scratch the requisite components: fast finite field arithmetic, elliptic-curve group arithmetic, pairing-based checks, and so on. Our elliptic curve choices, algorithm choices, and code are optimized and tailored for the the specific zk-SNARK protocol, with a focus on reducing proof verification time.

In our implementation we provide a choice between two security levels: 80 bits of security from an instantiation based on Edwards curves [Edw07]; or 128 bits of security from an instantiation based on Barreto–Naehrig curves [BN06] (taking the implementation of [Mit13] as a starting point). We emphasize that, while we present a concrete implementation, the techniques we employ can be applied to different algebraic setups, i.e., other choices of elliptic curves and security levels.

Proof verification enjoys excellent efficiency: at 80 bits of security, for an n-kilobyte input to the circuit, verification takes $\approx 4.7\,\mathrm{ms} + n \cdot (0.4\,\mathrm{ms})$, *regardless of circuit size*; at 128 bits of security, it takes $\approx 5.2\,\mathrm{ms} + n \cdot (0.5\,\mathrm{ms})$. The constant term (4.7 ms or 5.2 ms), which dominates for small inputs, corresponds to the verifier’s pairing-based checks. Notably, in both cases, this constant term is *less than half* the time for separately evaluating the 12 requisite pairings of the checks; we achieve this saving by merging parts of the pairings’ computation in a protocol-dependent way — another reason we need a custom implementation of the underlying math.

Our implementation also achieves good efficiency for key generation (i.e., for generating the proof system’s proving and verification keys) and proof generation. E.g., at 80 bits of security, for a 1-million-gate circuit C, generating proving and verification keys for C requires 97 s, and producing proofs of satisfiability for C requires 115 s; if C instead has 16 million gates, these numbers increase to 26 min and 30 min respectively.

Compared to previous implementations of zk-SNARKs for circuits [PGHR13, BCG$^+$13], our implementation is significantly more efficient; see Figure 2 for a comparison on a circuit with 1 million gates and a 10-field-element input.

	80 bits of security			128 bits of security		
	[BCG$^+$13]	**this work**	improvement	[PGHR13]	**this work**	improvement
Key generator	306 s	97 s	3.2×	123 s	117 s	1.1×
Prover	351 s	115 s	3.1×	784 s	147 s	5.3×
Verifier	66.1 ms	4.8 ms	13.8×	9.3 ms	5.7 ms	1.6×

Figure 2: Comparison with prior zk-SNARKs for a 1-million-gate arithmetic circuit and a 10-field-element input. (The comparison was conducted on our benchmarking machine, running software provided by the respective authors.)

As in prior work, space quickly becomes a bottleneck to key generation and proving. In our implementation, we leverage the *sparsity* of the QAP corresponding to a circuit so to shrink the size of public parameters; for instance, for the circuits generated by our circuit generator (discussed next), we obtain a 35% improvement in space.

Overall system: zk-SNARKs for a von Neumann architecture. By combining the above contributions, we obtain a zk-SNARK for proving/verifying correctness of vnTinyRAM computations; see Figure 3 and Figure 4. We evaluated our system for programs with up to 10,000 instructions, running for up to 32,000 steps. Verification time is only a few milliseconds, independent of the running time of the vnTinyRAM program, even when program size and input size are kilobytes. Proofs are always of constant size (230 bytes with 80 bits of security, or 288 bytes with 128 bits of security). Moreover, the efficiency of key generation and proving is significantly improved over prior works. For example, at 80 bits of security, for a 1,000-instruction program, given a 100-word input, and running for 10,000 steps: key generation takes 16.5 min; proving time is 5.4 min; and verification time is merely 4.8 ms.

Our contributions push the envelope for zk-SNARK implementations, by enriching the set of supported programs and improving efficiency. Yet, more work remains to be done to further slash costs of key generation and proving, so to achieve scalability to much longer computations (e.g., millions of vnTinyRAM cycles and beyond).

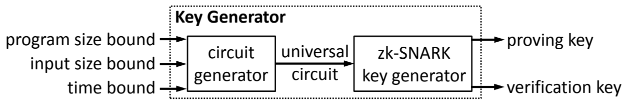

Figure 3: **Offline phase (once).** The key generator outputs a proving key and verification key, for proving and verifying correctness of any program execution meeting the given bounds.

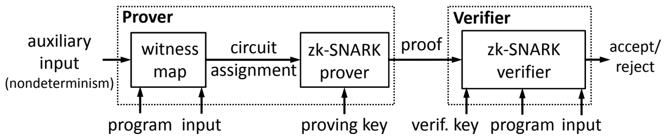

Figure 4: **Online phase (any number of times).** The prover sends a short and easy-to-verify proof to a verifier. This can be repeated any number of times, each time for a different program and input.

JIT case study: efficient `memcpy`. Besides evaluating our zk-SNARK for vnTinyRAM, we also provide an example to demonstrate the rich functionality supported by our system. We wrote a vnTinyRAM implementation of memcpy that leverages *just-in-time compilation* (in particular, *dynamic loop unrolling*) to improve cycle-count efficiency by $1.4\times$ (see Section 6.5). While ours is a simple case study, just-in-time compilation is a widely-used powerful technique that has found many applications, e.g., increasing the performance of interpreted programming languages such as Javascript in web browsers [GES$^+$09] or Python [RP06]. As the efficiency of zk-SNARK implementations improves, more and more of these applications will become feasible.

1.5 Roadmap

In Section 2 we provide preliminaries and background. In Section 3 we summarize vnTinyRAM and provide definitions of a zk-SNARK and circuit generator. In Section 4 we describe our circuit generator for vnTinyRAM. In Section 5 we summarize the optimizations of our zk-SNARK for circuit satisfiability. In Section 6 we evaluate our system, by performing microbenchmarks on primitive operations, estimating the costs of the two components of our system, and providing end-to-end performance data. In Section 8 we conclude and suggest future directions.

2 Preliminaries & Background

2.1 Notations

We denote by $\mathbb{F}$ a finite field and $\mathbb{F}_n$ is the field of size n. Field elements are denoted with Greek letters (e.g. α, β, γ). We denote by $\mathbb{F}[z]$ the ring of univariate polynomials over $\mathbb{F}$, and by $\mathbb{F}^{\leq d}[z]$ the subring of polynomials of degree $\leq d$. Vectors are denoted by arrow-equipped letters (such as $\vec{a}$); their entries carry an index but not the arrow (e.g., a_1 or a_2).

We shall encounter several concrete finite fields: two prime fields $\mathbb{F}_r$ and $\mathbb{F}_q$; and the two field extensions $\mathbb{F}_{q^3}$ and $\mathbb{F}_{q^6}$. When working with the prime field of size p we identify its elements with integers modulo p. The primes r and q respectively have 181 and 183 bits (i.e., $\lceil \log_2 r \rceil = 181$).

2.2 Arithmetic circuits

The circuits that we consider are not boolean but *arithmetic*. Given a finite field $\mathbb{F}$, an $\mathbb{F}$-*arithmetic circuit* takes inputs that are elements in $\mathbb{F}$, and its gates output elements in $\mathbb{F}$. We naturally associate a circuit with the function it computes. The circuits we consider only have *bilinear gates*.[2] Arithmetic circuit satisfiability is analogous to the boolean case:

Definition 2.1. *Let n, h, l respectively denote the input, witness, and output size. The* **circuit satisfaction problem** *of a circuit $C \colon \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$ with bilinear gates is defined by the relation $\mathcal{R}_C = \{(\vec{x}, \vec{a}) \in \mathbb{F}^n \times \mathbb{F}^h : C(\vec{x}, \vec{a}) = 0^l\}$;[3] its language is $\mathcal{L}_C = \{\vec{x} \in \mathbb{F}^n : \exists\, \vec{a} \in \mathbb{F}^h,\ C(\vec{x}, \vec{a}) = 0^l\}$.*

Our circuit generator reduces the correctness of program executions to arithmetic circuit satisfiability (see Section 4), and our zk-SNARK implementation produces/verifies proofs for this language (see Section 5).

All the arithmetic circuits we consider are over the prime field $\mathbb{F}_r$. In this case, when passing boolean strings as inputs to arithmetic circuits, we *pack* the string's bits into as few field elements as possible: given $x \in \{0,1\}^m$, we use $[\![x]\!]_r^m$ to denote the vector $\vec{x} \in \mathbb{F}_r^{|m|_r}$, where $|m|_r := \lceil m / \log r \rceil$, such that the binary representation of $x_i \in \mathbb{F}_r$ is the i-th block of $\log r$ bits in x (padded with 0's if needed). We also extend the notation $[\![x]\!]_r^m$ to binary strings $x \in \{0,1\}^n$ with $n < m$ bits via padding: $[\![x]\!]_r^m := [\![x0^{m-n}]\!]_r^m$.

2.3 Quadratic arithmetic programs

Our zk-SNARK implementation uses *quadratic arithmetic programs* (QAPs), introduced by Gennaro et al. [GGPR13].

Definition 2.2. *A* **quadratic arithmetic program** *of size m and degree d over $\mathbb{F}$ is a tuple $(\vec{A}, \vec{B}, \vec{C}, Z)$, where:*
- $\vec{A} = \big(A_0, A_1, \dots, A_m\big)$ *with* $A_i \in \mathbb{F}^{\leq d-1}[z]$;
- $\vec{B} = \big(B_0, B_1, \dots, B_m\big)$ *with* $B_i \in \mathbb{F}^{\leq d-1}[z]$;
- $\vec{C} = \big(C_0, C_1, \dots, C_m\big)$ *with* $C_i \in \mathbb{F}^{\leq d-1}[z]$;
- $Z \in \mathbb{F}[z]$ *has degree exactly d.*

Like a circuit, a QAP induces a satisfaction problem:

[2]A gate with inputs $x_1, \dots, x_m \in \mathbb{F}$ is *bilinear* if the output is $\langle \vec{a}, (1, x_1, \dots, x_m)\rangle \cdot \langle \vec{b}, (1, x_1, \dots, x_m)\rangle$ for some $\vec{a}, \vec{b} \in \mathbb{F}^{m+1}$. In particular, these include addition, multiplication, and constant gates.

[3]Throughout this paper, we identify a circuit (which is a directed acyclic graph with labeled vertices) with the function it computes.

Definition 2.3. *The* **satisfaction problem** *of a size-m QAP $(\vec{A},\vec{B},\vec{C},Z)$ is the relation $\mathcal{R}_{(\vec{A},\vec{B},\vec{C},Z)}$ of pairs $(\vec{x},\vec{s})$ such that*

- *$\vec{x} \in \mathbb{F}^n$, $\vec{s} \in \mathbb{F}^m$, and $n \le m$;*
- *$x_i = s_i$ for $i \in [n]$ (i.e., $\vec{s}$ extends $\vec{x}$); and*
- *the polynomial $Z(z)$ divides the following one*

$$(A_0(z) + \textstyle\sum_{i=1}^m s_i A_i(z)) \cdot (B_0(z) + \sum_{i=1}^m s_i B_i(z)) - (C_0(z) + \sum_{i=1}^m s_i C_i(z)) \ .$$

We denote by $\mathcal{L}_{(\vec{A},\vec{B},\vec{C},Z)}$ the language of $\mathcal{R}_{(\vec{A},\vec{B},\vec{C},Z)}$.

Gennaro et al. [GGPR13] showed that arithmetic circuit satisfiability can be efficiently reduced to the satisfiability of QAPs:

Lemma 2.4. *There exist two polynomial-time algorithms* QAPinst, QAPwit *with the following properties.*

- EFFICIENCY. *For any circuit $C\colon \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$ with α wires and β gates, $(\vec{A},\vec{B},\vec{C},Z) :=$ QAPinst(C) is a QAP of size m and degree d over $\mathbb{F}$, with $m = \alpha$ and $d = \beta + l$.*

Now fix a circuit C, and $(\vec{A},\vec{B},\vec{C},Z) :=$ QAPinst(C).

- CORRECTNESS. *For any $(\vec{x},\vec{a}) \in \mathcal{R}_C$, it holds that $(\vec{x},\vec{s}) \in \mathcal{R}_{(\vec{A},\vec{B},\vec{C},Z)}$, where $\vec{s} :=$ QAPwit$(C,\vec{x},\vec{a})$.*
- PROOF OF KNOWLEDGE. *For any $(\vec{x},\vec{s}) \in \mathcal{R}_{(\vec{A},\vec{B},\vec{C},Z)}$, it holds that $(\vec{x},\vec{a}) \in \mathcal{R}_C$, where $\vec{a}$ is a prefix of $\vec{s}$.*

Efficient zk-SNARK constructions for arithmetic circuit satisfiability rely on the above lemma to obtain a QAP corresponding to a circuit, and then utilize cryptographic techniques to prove/verify the QAP's satisfiability.

The intuition behind Lemma 2.4 is the following. The third condition in Definition 2.3 implies that $\langle 1 \circ \vec{s}, \vec{A}(\omega)\rangle \cdot \langle 1 \circ \vec{s}, \vec{B}(\omega)\rangle = \langle 1 \circ \vec{s}, \vec{C}(\omega)\rangle$ for all roots ω of Z. In other words, membership in $\mathcal{R}_{(\vec{A},\vec{B},\vec{C},Z)}$ is characterized by $\deg Z = d$ rank-1 quadratic constraints in the variable $\vec{s}$. By suitably selecting coefficients for the polynomials $\vec{A},\vec{B},\vec{C}$, one can encode satisfiability of an arithmetic circuit C into such constraints.

2.4 Pairings

We discuss zk-SNARK constructions in the next subsection (Section 2.5). However, before that, we need to introduce notation for pairings, which are used for verifying proofs.

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two cyclic groups of order r. We denote elements of $\mathbb{G}_1, \mathbb{G}_2$ via calligraphic letters such as $\mathcal{P}$, $\mathcal{Q}$. We write $\mathbb{G}_1$ and $\mathbb{G}_2$ in additive notation. Let $\mathcal{P}_1$ be a generator of $\mathbb{G}_1$, i.e., $\mathbb{G}_1 = \{\alpha\mathcal{P}_1\}_{\alpha\in\mathbb{F}_r}$ (α is also viewed as an integer, hence $\alpha\mathcal{P}_1$ is well-defined); let $\mathcal{P}_2$ be a generator for $\mathbb{G}_2$.

A *pairing* is an efficient map $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_T$ is also a cyclic group of order r (which we write in multiplicative notation), satisfying the following properties:

- BILINEARITY. For every nonzero elements $\alpha, \beta \in \mathbb{F}_r$, it holds that $e(\alpha\mathcal{P}_1, \beta\mathcal{P}_2) = e(\mathcal{P}_1, \mathcal{P}_2)^{\alpha\beta}$.
- NON-DEGENERACY. $e(\mathcal{P}_1, \mathcal{P}_2)$ is not the identity in $\mathbb{G}_T$.

Typically, $\mathbb{G}_1$ is instantiated as the group of points of an elliptic curve E defined over $\mathbb{F}_q$; $\mathbb{G}_2$ as the group of points of a *twist* of E; and $\mathbb{G}_T$ as an order-r subgroup of $\mathbb{F}^*_{q^k}$, where k is known as the *embedding degree* of the curve E.

For the high-level discussion of zk-SNARKs in the next subsection, the choice of instantiation of $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$, as well as the choice of pairing e, does not matter. However, later, when discussing optimizations in our implementation (see Section 5), these choices will matter a great deal.

2.5 zk-SNARK constructions

There are several constructions of zk-SNARKs in the literature [Gro10a, Lip12, GGPR13, BCI$^+$13, PGHR13, BCG$^+$13, Lip13]. The most efficient constructions are based either on *quadratic span programs* (QSPs) [GGPR13, Lip13] or *quadratic span programs* (QAPs) [GGPR13, BCI$^+$13, PGHR13, BCG$^+$13]. While QSPs allow for tight reductions from boolean circuits, QAPs allow for tight reductions from arithmetic ones (see Lemma 2.4).

In this work, we consider arithmetic circuits, because arithmetic gates significantly improve the efficiency of our circuit generator (see Section 4). Thus, we focus on zk-SNARKs based on QAPs. Our starting point is the zk-SNARK for arithmetic circuit satisfiability of Parno et al. [PGHR13]. The protocol is summarized in Figure 5. As mentioned in Section 1 and discussed in Section 5, one of our contributions is a *high-performance* implementation of this protocol, to be used for the circuits produced by our circuit generator.

We refer to [PGHR13] for additional details regarding the intuition for the protocol, as well as the cryptographic assumptions on which its proof of security relies. (Briefly, security relies on the q-power Diffie-Hellman, q-power knowledge-of-exponent, and q-strong Diffie-Hellman assumptions [Gro10b, BB04, Gen04] for q that depends polynomially on the arithmetic circuit's size.)

3 zk-SNARKs for a Simple von Neumann RISC Architecture

In Section 3.1 we summarize the main features of vnTinyRAM, a simple von Neumann RISC architecture we introduce. In Section 3.2, we informally define zk-SNARKs for vnTinyRAM. In Section 3.3 we define the notion of a circuit generator for vnTinyRAM. In Section 3.4 we explain how to obtain a zk-SNARK for vnTinyRAM by combining a circuit generator for vnTinyRAM and a zk-SNARK for arithmetic circuit satisfiability.

3.1 A summary of vnTinyRAM

Ben-Sasson et al. [BCG$^+$13] introduced TinyRAM: a RISC architecture, with word-addressable memory, following the Harvard paradigm. We modify TinyRAM to obtain vnTinyRAM, which differs from it in two main ways.

- vnTinyRAM follows the von Neumann paradigm, whereby program and data are stored in the same read-write address space. Programs may use *runtime code generation*.
- vnTinyRAM has byte-addressable memory, along with instructions to load/store bytes and to load/store words. Byte accesses are typically used by programs performing array or string operations, while word accesses by programs performing arithmetic. Simultaneous support for both greatly simplifies compiling to vnTinyRAM.

Besides the above two main differences, vnTinyRAM is very similar to TinyRAM. Due to space limitations, here we limit our discussion to only a high-level summary containing just enough details for discussing our circuit generator later on.

Overview of vnTinyRAM. vnTinyRAM is parametrized by the *word size*, denoted W and chosen to be a power of 2 and divisible by 8, and the *number of registers*, denoted K.

The *state* of the machine consists of the following.

- The *program counter*; it consists of W bits.

Public parameters. A prime r, two cyclic groups $\mathbb{G}_1$ and $\mathbb{G}_2$ of order r with generators $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively, and a pairing $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ (where $\mathbb{G}_T$ is also cyclic of order r).

(a) Key generator G

- INPUTS: circuit $C\colon \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^l$
- OUTPUTS: proving key pk and verification key vk

1. Compute $(\vec{A}, \vec{B}, \vec{C}, Z) = \mathsf{QAPinst}(C)$; extend $\vec{A}, \vec{B}, \vec{C}$ via
$$A_{m+1} = B_{m+2} = C_{m+3} = Z \,,$$
$$A_{m+2} = A_{m+3} = B_{m+1} = B_{m+3} = C_{m+1} = C_{m+2} = 0 \,.$$
2. Randomly sample $\tau, \rho_{\mathsf{A}}, \rho_{\mathsf{B}}, \alpha_{\mathsf{A}}, \alpha_{\mathsf{B}}, \alpha_{\mathsf{C}}, \beta, \gamma \in \mathbb{F}_r$.
3. Set $\mathsf{pk} := (C, \mathsf{pk}_{\mathsf{A}}, \mathsf{pk}'_{\mathsf{A}}, \mathsf{pk}_{\mathsf{B}}, \mathsf{pk}'_{\mathsf{B}}, \mathsf{pk}_{\mathsf{C}}, \mathsf{pk}'_{\mathsf{C}}, \mathsf{pk}_{\mathsf{K}}, \mathsf{pk}_{\mathsf{H}})$ where for $i = 0,1,\ldots,m+3$:
$$\mathsf{pk}_{\mathsf{A},i} := A_i(\tau)\rho_{\mathsf{A}}\mathcal{P}_1 \,, \quad \mathsf{pk}'_{\mathsf{A},i} := A_i(\tau)\alpha_{\mathsf{A}}\rho_{\mathsf{A}}\mathcal{P}_1 \,,$$
$$\mathsf{pk}_{\mathsf{B},i} := B_i(\tau)\rho_{\mathsf{B}}\mathcal{P}_2 \,, \quad \mathsf{pk}'_{\mathsf{B},i} := B_i(\tau)\alpha_{\mathsf{B}}\rho_{\mathsf{B}}\mathcal{P}_1 \,,$$
$$\mathsf{pk}_{\mathsf{C},i} := C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}\mathcal{P}_1 \,, \mathsf{pk}'_{\mathsf{C},i} := C_i(\tau)\alpha_{\mathsf{C}}\rho_{\mathsf{A}}\rho_{\mathsf{B}}\mathcal{P}_1 \,,$$
$$\mathsf{pk}_{\mathsf{K},i} := \beta\big(A_i(\tau)\rho_{\mathsf{A}} + B_i(\tau)\rho_{\mathsf{B}} + C_i(\tau)\rho_{\mathsf{A}}\rho_{\mathsf{B}}\big)\mathcal{P}_1 \,,$$
and for $i = 0,1,\ldots,d$, $\mathsf{pk}_{\mathsf{H},i} := t^i\mathcal{P}_1$.
4. Set $\mathsf{vk} := (\mathsf{vk}_{\mathsf{A}}, \mathsf{vk}_{\mathsf{B}}, \mathsf{vk}_{\mathsf{C}}, \mathsf{vk}_{\gamma}, \mathsf{vk}^1_{\beta\gamma}, \mathsf{vk}^2_{\beta\gamma}, \mathsf{vk}_{\mathsf{Z}}, \mathsf{vk}_{\mathsf{IC}})$ where
$$\mathsf{vk}_{\mathsf{A}} := \rho_{\mathsf{A}}\alpha_{\mathsf{A}}\mathcal{P}_2 \,, \mathsf{vk}_{\mathsf{B}} := \rho_{\mathsf{B}}\alpha_{\mathsf{B}}\mathcal{P}_1 \,, \mathsf{vk}_{\mathsf{C}} := \rho_{\mathsf{A}}\rho_{\mathsf{B}}\alpha_{\mathsf{C}}\mathcal{P}_2$$
$$\mathsf{vk}_{\gamma} := \gamma\mathcal{P}_2 \,, \quad \mathsf{vk}^1_{\beta\gamma} := \gamma\beta\mathcal{P}_1 \,, \quad \mathsf{vk}^2_{\beta\gamma} := \gamma\beta\mathcal{P}_2 \,,$$
$$\mathsf{vk}_{\mathsf{Z}} := Z(\tau)\rho_{\mathsf{C}}\mathcal{P}_2 \,, \big(\mathsf{vk}_{\mathsf{IC},i}\big)_{i=0}^n := \big(A_i(\tau)\mathcal{P}_1\big)_{i=0}^n \,.$$
5. Output $(\mathsf{pk}, \mathsf{vk})$.

Key sizes. When invoked on a circuit $C\colon \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^l$ with a wires and b (bilinear) gates, the key generator outputs:

- pk with $(6a + b + l + 25)$ $\mathbb{G}_1$-elements and $(a + 4)$ $\mathbb{G}_2$-elements;
- vk with $(n + 3)$ $\mathbb{G}_1$-elements and 5 $\mathbb{G}_2$-elements.

Proof size The proof always has 7 $\mathbb{G}_1$-elements and 1 $\mathbb{G}_2$-element.

(b) Prover P

- INPUTS: proving key pk, input $\vec{x} \in \mathbb{F}_r^n$, and witness $\vec{a} \in \mathbb{F}_r^h$
- OUTPUTS: proof π

1. Compute $(\vec{A}, \vec{B}, \vec{C}, Z) := \mathsf{QAPinst}(C)$.
2. Compute $\vec{s} := \mathsf{QAPwit}(C, \vec{x}, \vec{a}) \in \mathbb{F}_r^m$.
3. Randomly sample $\delta_1, \delta_2, \delta_3 \in \mathbb{F}_r$.
4. Compute $\vec{h} = (h_0, h_1, \ldots, h_d) \in \mathbb{F}_r^{d+1}$, which are the coefficients of $H(z) := \frac{A(z)B(z)-C(z)}{Z(z)}$ where $A, B, C \in \mathbb{F}_r[z]$ are as follows:
$$A(z) := A_0(z) + \textstyle\sum_{i=1}^m s_i A_i(z) + \delta_1 Z(z) \,,$$
$$B(z) := B_0(z) + \textstyle\sum_{i=1}^m s_i B_i(z) + \delta_2 Z(z) \,,$$
$$C(z) := C_0(z) + \textstyle\sum_{i=1}^m s_i C_i(z) + \delta_3 Z(z) \,.$$
5. Letting $\vec{c} := (1 \circ \vec{s} \circ \delta_1 \circ \delta_2 \circ \delta_3) \in \mathbb{F}_r^{4+m}$, compute
$$\pi_{\mathsf{A}} := \langle \vec{c}, \mathsf{pk}_{\mathsf{A}} \rangle,\ \pi'_{\mathsf{A}} := \langle \vec{c}, \mathsf{pk}'_{\mathsf{A}} \rangle,\ \pi_{\mathsf{B}} := \langle \vec{c}, \mathsf{pk}_{\mathsf{B}} \rangle,\ \pi'_{\mathsf{B}} := \langle \vec{c}, \mathsf{pk}'_{\mathsf{B}} \rangle,$$
$$\pi_{\mathsf{C}} := \langle \vec{c}, \mathsf{pk}_{\mathsf{C}} \rangle,\ \pi'_{\mathsf{C}} := \langle \vec{c}, \mathsf{pk}'_{\mathsf{C}} \rangle,\ \pi_{\mathsf{K}} := \langle \vec{c}, \mathsf{pk}_{\mathsf{K}} \rangle,\ \pi_{\mathsf{H}} := \langle \vec{h}, \mathsf{pk}_{\mathsf{H}} \rangle.$$
6. Output $\pi := (\pi_{\mathsf{A}}, \pi'_{\mathsf{A}}, \pi_{\mathsf{B}}, \pi'_{\mathsf{B}}, \pi_{\mathsf{C}}, \pi'_{\mathsf{C}}, \pi_{\mathsf{K}}, \pi_{\mathsf{H}})$.

(c) Verifier V

- INPUTS: verification key vk, input $\vec{x} \in \mathbb{F}_r^n$, and proof π
- OUTPUTS: decision bit

1. Compute $\mathsf{vk}_{\vec{x}} := \mathsf{vk}_{\mathsf{IC},0} + \sum_{i=1}^n x_i \mathsf{vk}_{\mathsf{IC},i} \in \mathbb{G}_1$.
2. Check validity of knowledge commitments for A, B, C:
$$e(\pi_{\mathsf{A}}, \mathsf{vk}_{\mathsf{A}}) = e(\pi'_{\mathsf{A}}, \mathcal{P}_2) \,, e(\mathsf{vk}_{\mathsf{B}}, \pi_{\mathsf{A}}) = e(\pi'_{\mathsf{B}}, \mathcal{P}_2) \,,$$
$$e(\pi_{\mathsf{C}}, \mathsf{vk}_{\mathsf{C}}) = e(\pi'_{\mathsf{C}}, \mathcal{P}_2) \,.$$
3. Check same coefficients were used:
$$e(\pi_{\mathsf{K}}, \mathsf{vk}_{\gamma}) = e(\mathsf{vk}_{\vec{x}} + \pi_{\mathsf{A}} + \pi_{\mathsf{C}}, \mathsf{vk}^2_{\beta\gamma}) \cdot e(\mathsf{vk}^1_{\beta\gamma}, \pi_{\mathsf{B}}) \,.$$
4. Check QAP divisibility:
$$e(\mathsf{vk}_{\vec{x}} + \pi_{\mathsf{A}}, \pi_{\mathsf{B}}) = e(\pi_{\mathsf{H}}, \mathsf{vk}_{\mathsf{Z}}) \cdot e(\pi_{\mathsf{C}}, \mathcal{P}_2) \,.$$
5. Accept if and only if all the above checks succeeded.

Figure 5: **zk-SNARK for arithmetic circuit satisfiability.** One of our contributions is a *high-performance* implementation of the above protocol, which is from Parno et al. [PGHR13]. The resulting zk-SNARK can be used to prove/verify satisfiability of circuits output by our circuit generator.

- K general-purpose *registers*; each consists of W bits.
- The *(condition) flag*, denoted flag; it consists of 1 bit.
- *Memory*, which is a linear array of 2^W bytes.
- Two *tapes*, each with a string of W-bit words, and read-only in one direction. One tape is for a *primary input* x and the other for an *auxiliary input* w. We treat the primary input as given, and the auxiliary input

as nondeterministic.

At each step, the machine executes an *instruction*, which changes the machine's state. Briefly, the instruction set of vnTinyRAM includes load and store instructions for accessing random-access memory (in byte or word blocks), as well as simple integer, shift, logical, compare, move, and jump instructions. Thus, vnTinyRAM can efficiently implement control flow, loops, subroutines, recursion, and so on. Complex instructions (e.g., floating-point arithmetic) are not directly supported and can be implemented "in software".

In memory, any vnTinyRAM instruction is represented as a double word. A *program* $\mathbf{P}$ is a list of address/double-word pairs specifying the initial contents of memory; all other memory locations assume the initial value of 0. Thus, the *initial state* of the machine is as follows. The contents of all general-purpose registers and flag are all 0. The content of one tape defines the primary input and that of the other tape defines the auxiliary input. The initial contents of pc is 0, so that execution begins with the instruction encoded by the double word aligned to the 0-th byte in memory.

At every time step, the machine executes the instruction encoded by the double word aligned to the $[\mathsf{pc}]_{2W}$-th byte in memory, where $[\mathsf{pc}]_{2W}$ denotes the unsigned integer given by pc rounded down to a multiple of $2W/8$; every instruction increments pc by $2W/8$, unless it explicitly modifies pc. (Note that $2W/8$ is the number of bytes in a double word.)

The machine's only input is via the input tapes and initial memory, and only output is via an `answer` instruction (which halts execution) having a single argument A, representing the return value. By default, $A = 0$ means "accept".

Parameter choices. Two examples of natural parameter choices are $W, K = 16$ (a 16-bit machine with 16 registers) and $W, K = 32$ (a 32-bit machine with 32 registers). For concreteness, in the rest of the paper we fix $W, K = 16$.

3.2 Informal definition of zk-SNARK for vnTinyRAM

At high-level, a zk-SNARK for vnTinyRAM is a cryptographic primitive that gives short and easy-to-verify non-interactive zero-knowledge proofs of knowledge for the correct execution of programs on the machine vnTinyRAM.

We only provide an informal definition, and refer to prior work for a formal definition of zk-SNARKs for random-access machines [BCI+13]. Below, the security parameter is implicit.

We begin by introducing the following useful notation:

Definition 3.1. *Fix bounds ℓ, n, T. The language $\mathcal{L}_{\ell,n,T}$ consists of pairs $(\mathbf{P}, x)$ such that:*
- *$\mathbf{P}$ is a program with $\leq \ell$ instructions,*
- *x is a primary input with $\leq n$ words,*
- *There exists an auxiliary input w s.t. $\mathbf{P}(x, w)$ accepts in $\leq T$ steps.*

We denote by $\mathcal{R}_{\ell,n,T}$ the relation corresponding to $\mathcal{L}_{\ell,n,T}$.

A **zk-SNARK for** vnTinyRAM is a triple of polynomial-time algorithms $(\mathsf{KeyGen}, \mathsf{Prove}, \mathsf{Verify})$ working as follows.

- $\mathsf{KeyGen}(\ell, n, T) \to (\mathsf{pk}, \mathsf{vk})$. On input a program size bound ℓ, time bound T, and primary-input size bound n, the *key generator* KeyGen probabilistically samples a proving key pk and a verification key vk.

The keys pk and vk are published as public parameters and can be used, any number of times, to prove and verify membership in the language $\mathcal{L}_{\ell,n,T}$ as follows.

- $\mathsf{Prove}(\mathsf{pk}, \mathbf{P}, x, w) \to \pi$. On input a program $\mathbf{P}$ with $\leq \ell$ instructions, x with $\leq n$ words, and w such that $\mathbf{P}(x, w)$ accepts in $\leq T$ steps, the *prover* Prove outputs a non-interactive proof π for the statement $(\mathbf{P}, x) \in \mathcal{L}_{\ell,n,T}$.
- $\mathsf{Verify}(\mathsf{vk}, \mathbf{P}, x, \pi) \to b$. On input a program $\mathbf{P}$ with $\leq \ell$ instructions and x with $\leq n$ words, the *verifier* Verify outputs $b = 1$ if he is convinced that $(\mathbf{P}, x) \in \mathcal{L}_{\ell,n,T}$.

The key generator KeyGen is "universal": it does not depend on the program $\mathbf{P}$ or primary input x, but only on their respective size bounds ℓ and n (as well as the time bound T).

A zk-SNARK satisfies the following properties.

Correctness. The honest prover can convince the verifier for any instance in the language. I.e., for every $(\mathbf{P}, x) \in \mathcal{L}_{\ell,n,T}$ with a witness w,

$$\Pr\left[\mathsf{Verify}(\mathsf{vk}, \mathbf{P}, x, \pi) = 1 \;\middle|\; \begin{array}{r} (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\ell, n, T) \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pk}, \mathbf{P}, x, w) \end{array}\right] = 1 \;.$$

Succinctness. An honestly-generated proof π has $O(1)$ bits, and $\mathsf{Verify}(\mathsf{vk}, \mathbf{P}, x, \pi)$ runs in time $O(\ell + n)$. In particular, verification time does *not* depend on the time bound T.

Proof of knowledge (& soundness). If the verifier accepts a proof, the prover "knows" a witness for the instance. (Thus, soundness holds.) I.e., for every polynomial-size adversary A there is a polynomial-size witness extractor E s.t.

$$\Pr\left[\begin{array}{c} \mathsf{Verify}(\mathsf{vk}, \mathbf{P}, x, \pi) = 1 \\ \big((\mathbf{P}, x), w\big) \notin \mathcal{R}_{\ell,n,T} \end{array} \;\middle|\; \begin{array}{r} (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\ell, n, T) \\ (\mathbf{P}, x, \pi) \leftarrow A(\mathsf{pk}, \mathsf{vk}) \\ w \leftarrow E(\mathsf{pk}, \mathsf{vk}) \end{array}\right] \leq \mathsf{negl} \;.$$

Zero knowledge. The proof π is statistical zero knowledge.

3.3 Definition of circuit generator for vnTinyRAM

As we discuss in the next subsection (Section 3.4), one of the components of our zk-SNARK for vnTinyRAM is a circuit generator for vnTinyRAM; its purpose is to produce circuits that verify execution of vnTinyRAM programs. We now define the notion of a circuit generator. Below, we use the notations $[\![\cdot]\!]_r$ and $|\cdot|_r$, introduced in Section 2.2.

Definition 3.2. *A* **circuit generator** *of efficiency $f(\cdot)$ is a polynomial-time algorithm* circ*, together with an efficient* witness map wit*, with the following properties.*

- Efficiency. *For any program size bound ℓ, time bound T, and primary-input size bound n, $C := \mathsf{circ}(\ell, n, T)$ is an $f(\ell, n, T)$-gate $\mathbb{F}_r$-arithmetic circuit $C \colon \mathbb{F}_r^m \times \mathbb{F}_r^h \to \mathbb{F}_r^l$, for $m := |\ell 2W|_r + |nW|_r$ and some h, l, where W is the word size (cf. Section 3.1).*

Now fix bounds ℓ, n, T, and let $C := \mathsf{circ}(\ell, n, T)$.

- Correctness. *For every $(\mathbf{P}, x) \in \mathcal{L}_{\ell,n,T}$ with witness w, it holds that $(\vec{x}, \vec{a}) \in \mathcal{R}_C$, where $\vec{x} := [\![\mathbf{P}]\!]_r^{\ell 2W} \circ [\![x]\!]_r^{nW}$ and $\vec{a} := \mathsf{wit}(\ell, n, T, \mathbf{P}, x, w)$.*
- Proof of knowledge. *Given any $(\vec{x}, \vec{a}) \in \mathcal{R}_C$, one can efficiently "extract" a witness w for $(\mathbf{P}, x) \in \mathcal{L}_{\ell,n,T}$.*

```
KeyGen
• INPUTS: bounds $\ell, n, T$
• OUTPUTS: proving key $\mathsf{pk}$ and verification key $\mathsf{vk}$
1. Compute $C := \mathsf{circ}(\ell, n, T)$.
2. Compute $(\mathsf{pk}, \mathsf{vk}) := G(C)$, and output $(\mathsf{pk}, \mathsf{vk})$.
Prove
• INPUTS: proving key $\mathsf{pk}$ and $(\mathbf{P}, x) \in \mathcal{L}_{\ell,n,T}$ with witness $w$
• OUTPUTS: proof $\pi$
1. Compute $\vec{x} := [\![\mathbf{P}]\!]_r^{\ell 2W} \circ [\![x]\!]_r^{nW}$.
2. Compute $\vec{a} := \mathsf{wit}(\ell, n, T, \mathbf{P}, x, w)$.
3. Compute $\pi := P(\mathsf{pk}, \vec{x}, \vec{a})$, and output $\pi$
Verify
• INPUTS: verification key $\mathsf{vk}$ and $(\mathbf{P}, x) \in \mathcal{L}_{\ell,n,T}$
• OUTPUTS: decision bit
1. Compute $\vec{x} := [\![\mathbf{P}]\!]_r^{\ell 2W} \circ [\![x]\!]_r^{nW}$.
2. Compute $b := V(\mathsf{vk}, \vec{x}, \pi)$, and output $b$.
```

Figure 6: Our circuit generator and our zk-SNARK for arithmetic circuit satisfiability can be combined to obtain a **zk-SNARK for** vnTinyRAM.

The circuit C output by circ does not depend on the program $\mathbf{P}$ or primary input x, but only on their respective size bounds ℓ and n (as well as the time bound T). This fact enables KeyGen to be *universal* (see Section 3.4). In particular, universality makes it possible to generate keys for all bound choices, in powers of 2 up to a certain constant, once and for all; afterwards, one can pick the key corresponding to bounds fitting a given computation.

3.4 Construction of a zk-SNARK for vnTinyRAM

We obtain a zk-SNARK for vnTinyRAM by combining:

- A **circuit generator for** vnTinyRAM. We use it to make circuits that verify execution of vnTinyRAM programs.
- A **zk-SNARK for arithmetic circuit satisfiability**. We use it to produce and verify proofs for the circuits output by the circuit generator for vnTinyRAM.

See Figure 6 for more details. We next proceed as follows. In Section 4 we summarize the construction of our circuit generator for vnTinyRAM. In Section 5 we summarize the optimizations in our implementation of the zk-SNARK for circuit satisfiability from Figure 5. In Section 6 we evaluate each component, as well as the combined system.

4 A Circuit Generator for a Simple von Neumann RISC Architecture

A circuit generator translates the correctness of *any* suitably-bounded program execution into circuit satisfiability. (See definition in Section 3.3.) We show:

Theorem 4.1. *For any program size bound ℓ, time bound T, and primary-input size bound n, there exists a circuit generator of efficiency*

$$f(\ell, n, T) := O\big((\ell + n + T) \cdot \log(\ell + n + T)\big) \enspace .$$

For comparison, the circuit generator in [BCG+13] has efficiency $\Theta\big((n + T) \cdot (\log(n + T) + \ell)\big)$. Thus, ours is a quadratic improvement over previous work: the size of our circuit does not grow with $\ell \cdot T$, but only with $\ell + T$. As our evaluation demonstrates (see Section 6.2), this improvement translates into significant savings in practice.

In short, previous work hardcoded the program into each step's verification. Instead, we store the program in memory, and we verify instruction fetch by relying on the same routing network that is also used for verifying data loads/stores.

4.1 Past techniques

Our results extends previous techniques, so we begin by recalling these techniques. One of the main difficulties that arise when designing a circuit generator is: *how to (efficiently!) handle memory accesses?* Accesses to memory depend on the inputs of the program, while a circuit's topology does not depend on its inputs. Thus, a circuit tasked with verifying program executions must be "ready" to support access patterns arising from any possible input.

Of course, we could restrict our focus only to programs that do not exhibit data dependencies [PGHR13]. Doing so simplifies the circuit generation problem, but also severely limits functionality. Indeed, most programs have data dependencies. Thus, the problem requires a solution.

The naive approach. A naive approach is for the circuit generator to construct a layered circuit that works as follows. The i-th layer contains the entire state of the machine (CPU state and random-access memory) at time step i, and layer $i + 1$ is computed from it by evaluating the transition function of the machine, handling any accesses to memory via multiplexing. However, this approach is very wasteful: for a program that accesses S addresses in memory, the resulting circuit has size that is at least $\Omega(TS)$. In general, S may be as large as T, so that the circuit has quadratic size.

An efficient approach: nondeterministic routing. Sorting and routing are ubiquitous techniques in fast simulation results between nondeterministic models of computation [Ofm65, Sch78, GS89, Rob91]. Ben-Sasson et al. [BCGT13] suggested using *nondeterministic routing* on a Beneš network as the basis for a circuit generator for random-access machines (RAMs). Ben-Sasson et al. [BCG+13] introduced a simple RAM architecture, called TinyRAM, and constructed a routing-based circuit generator for TinyRAM.

We summarize the main idea behind nondeterministic routing; for more details, see [BCG+13]. We introduce two notions:

- A *CPU state*, denoted S, is a string of $(W + KW + 1)$ bits, encoding the values of the program counter, K registers, and condition flag at a given time step.
- An *execution trace* for a program $\mathbf{P}$, time bound T, and primary input x is a sequence of states $\mathsf{tr} = (S_1, \ldots, S_T)$. An execution trace tr is *valid* if there is an auxiliary input w such that the sequence of states induced by $\mathbf{P}$ running with input tapes (x, w) is tr.

We seek an arithmetic circuit C for verifying that tr is valid. We break this down by splitting validity into 3 sub-properties:

- *Validity of instruction fetch.* For every time step, the correct instruction is fetched.
- *Validity of instruction execution.* For every time step, the fetched instruction is correctly executed (up to memory).
- *Validity of memory accesses.* Every load from an address retrieves the value of the last store to that address.

The first two properties are verified as follows. Construct a circuit $C_{\mathbf{P}}$ so that, for any two states S and S', $C_{\mathbf{P}}(S, S', g)$ is satisfied for some "guess" g if and only if S' can be reached from S (by fetching from $\mathbf{P}$ the instruction indicated by the program counter in S and then executing it), for *some* state of memory. Then, the two properties hold if $C_{\mathbf{P}}(S_i, S_{i+1}, \cdot)$ is satisfiable for $i = 1, \dots, T-1$. Thus, C contains T copies of $C_{\mathbf{P}}$, each wired to a pair of adjacent states in tr.

The third property is verified via nondeterministic routing. Assume that C also gets as input $\mathsf{MemSort}(\mathsf{tr})$, which is the *same* trace tr but sorted by accessed memory addresses (breaking ties via timestamps). One can write a circuit C_{mem} so that validity of memory accesses holds if C_{mem} is satisfied by each pair of adjacent states in $\mathsf{MemSort}(\mathsf{tr})$. (Roughly, C_{mem} checks for consistency of "load-after-load", "load-after-store", and so on.) But C *does not know* if the auxiliary input tr^* equals $\mathsf{MemSort}(\mathsf{tr})$. So C works as follows: (a) C has T copies of C_{mem}, each wired to a pair of adjacent states in tr^*; (b) C verifies that $\mathsf{tr}^* = \mathsf{MemSort}(\mathsf{tr})$ by routing on a $O(T \log T)$-node Beneš network. The switches of the routing network are set according to non-deterministic guesses and the routing network merely *verifies* that the switch settings induce a permutation. This allows for a particularly tight reduction: all known constructions that *compute* the correct permutation hide impractical constants in big-Oh notation [AKS83]. In fact, non-determinism is key ingredient thorough the reduction: it is used to get tight sub-circuits for multiplexing, division (by guessing the result and verify it by multiplication), etc.

Past inefficiencies. After filling in additional details, the construction of [BCG+13] summarized above gives a circuit of size:

$$\Theta\big((n+T) \cdot (\log(n+T) + \ell)\big) = \Omega(\ell \cdot T) \ .$$

The $\Omega(\ell \cdot T)$ arises from the fact that the circuit $C_{\mathbf{P}}$ has the program $\mathbf{P}$ *hardcoded*, so that $C_{\mathbf{P}}$ has size $\Omega(\ell)$ for ℓ-instruction programs. When ℓ is already in thousands, the size of $C_{\mathbf{P}}$ is completely dominated by program size, and the corresponding cost of $\Omega(\ell T)$ does not scale.

4.2 Our construction

We summarize how our more efficient circuit generator works. We do not hardcode the program $\mathbf{P}$ in a circuit that is repeated T times. Instead, our machine follows the von Neumann paradigm: $\mathbf{P}$ lies in the same read-write memory as data. But then how do we verify validity of instruction fetch when an instruction is loaded from memory?

At the highest level, we do not distinguish between accesses to memory caused by instruction fetch or load/store instructions, and verify consistency of both using the *same* nondeterministic routing network. (This would achieve both validity of instruction fetch *and* validity of memory accesses.)

1. We extend the execution trace to include instructions, besides CPU states. Namely, $\mathsf{tr} = (I_1, S_1, \dots, I_T, S_T)$ where I_i is the instruction executed at step i.

2. We extend the auxiliary input tr^* to (allegedly) contain the "memory sort" of *both* instructions and CPU states; sorting is verified by routing its $2T$ items on a Beneš network.

While the high level description is intuitive, a number of technical challenges need to be overcome.

First, byte-addressable memory is now accessed in different blocks: instruction-size blocks when loading instructions during instruction fetch; word-size blocks when loading/storing words; and byte-size blocks when loading/storing bytes. These different accesses complicate the task of C_{mem}, which is checking the validity of each pair of items in the memory-sorted tr^*. Our design of C_{mem} only has $200 + \log T$ gates.

Second, we need to ensure that, when the machine starts executing, the program $\mathbf{P}$ is *already* correctly loaded in memory. (It does not suffice for memory to be initialized to *some* values.) We solve this problem by extending the execution trace with an initial *boot* section, preceding the beginning of computation, during which the program is stored in memory, one instruction at a time. We then ensure correctness of the stores by using the same Beneš network, which now must route $\ell + 2T$ for ℓ-instruction programs.

The above description is only intuitive, and, due to space limitations, does not discuss all optimizations that ultimately yield the performance that we report in Section 6.2.

5 A High-Performance zk-SNARK For Arithmetic Circuit Satisfiability

We summarize the optimizations that we have used in our implementation of a zk-SNARK for arithmetic circuit satisfiability. (See Figure 5 for a high-level summary of the key generator, prover, and verifier algorithms.) The zk-SNARK can be used to produce and verify proofs for the circuits output by the circuit generator of Section 4.

As mentioned in Section 1.4, our zk-SNARK implementation provides a choice between two security levels: 80 bits or 128 bits. For simplicity, in this section (and later in Section 6) we only discuss the 80-bit security level. Nevertheless, most of the optimizations that we shall discuss are generic, and extend to other algebraic setups and security levels.

5.1 Fast finite field and curve arithmetic

The key generator, prover, and verifier perform many arithmetic operations, in various finite fields as well as over different elliptic curves. An efficient implementation of these three algorithms must first of all be built on fast arithmetic.

Montgomery reduction for prime fields. Arithmetic in fields of prime order is used for several tasks, including implementing other kinds of arithmetic. For example, the key generator performs $\mathbb{F}_r$-arithmetic when evaluating each $A_i, B_i, C_i \in \mathbb{F}_r[z]$ at $\tau \in \mathbb{F}_r$ (see Step 3 in Figure 5a), and the prover when computing the coefficients $\vec{h}$ of $H \in \mathbb{F}_r[z]$ (see Step 4 in Figure 5b). Moreover, $\mathbb{F}_q$-arithmetic underlies arithmetic in $\mathbb{G}_1$ and $\mathbb{G}_2$, as well as pairing computations.

We achieved highly-optimized implementations of arithmetic for $\mathbb{F}_r$ and $\mathbb{F}_q$ via *Montgomery reduction* [Mon85], which is a technique used for speeding up field multiplication. Indeed, field addition and subtraction are typically cheap operations: it suffices to add/subtract the representative field elements and adjust the result should it "wrap around". In contrast, field multiplication is more complex: adjusting the overflow after multiplication requires a *modular reduction*, which maps the result to its remainder modulo the prime.

The *Montgomery representation* of $a \in \mathbb{F}_p$ is the integer $[a] := aR \mod p$. Adding/subtracting Montgomery representations is the same as adding/subtracting in $\mathbb{F}_p$. Crucially, the reduction step $[a][b] \to [ab]$ ($abR^2 \to abR \mod p$) only requires a few simple operations and then dividing the result by R. If R is a power of 2, division by R corresponds to right-shifting, which permits particularly efficient implementation.

Following the SOS method outlined in [KAK96], our implementation of field multiplication does multiplication and Montgomery reduction simultaneously (interleaving the corresponding instructions). Compared to ordinary modulus reduction, we achieve a $4\times$ speedup.

Towering-friendly fields. As we shall discuss, arithmetic in $\mathbb{F}_{q^3}$ is used to implement operations in $\mathbb{G}_2$, while arithmetic in $\mathbb{F}_{q^6}$ is used in pairing computations by the verifier.

Because $q \equiv 1 \bmod 6$ for us, the field $\mathbb{F}_{q^6}$ is *towering friendly* [BS10], so that $\mathbb{F}_{q^6}$ can be represented via successive root extractions. In particular, following [DOhSD06], we construct $\mathbb{F}_{q^6}$ as $\mathbb{F}_{q^3}[Y]/(Y^2 - X)$ with $\mathbb{F}_{q^3} = \mathbb{F}_q[X]/(X^3 - \delta)$ and δ is not a square nor a cube in $\mathbb{F}_q$. This tower representation reduces the cost of squaring, multiplication, and Frobenius map (which sends α to α^q) in $\mathbb{F}_{q^6}$, all of which, as we discuss later, are important for pairing computations.

For the cubic and quadratic extensions, we use the Karatsuba method for squaring and multiplication [Knu97, DOhSD06], and *inversion via multiplication* [Knu97, LH00, BGDM$^+$10] for inversion.

Edwards curves. Recall from Section 2.4 that $\mathbb{G}_1$ and $\mathbb{G}_2$ are cyclic groups of order r that admit an efficient pairing $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. (The pairing allows the verifier to perform the necessary checks; see Figure 5c.) Thus, $\mathbb{G}_1$ and $\mathbb{G}_2$ need to be instantiated as groups defined over a *pairing-friendly elliptic curve*. Yet, since different instantiations yield different efficiency, the choice of instantiation must be done with care.

Our curve choice is the same as in [BCG$^+$13]: we use pairing-friendly *Edwards curves* [Edw07]. We briefly explain the rationale behind this choice. The complexity of the key generator and prover is dominated by the cost of arithmetic in $\mathbb{G}_1$ and $\mathbb{G}_2$; moreover, both perform many more operations in $\mathbb{G}_1$ than in $\mathbb{G}_2$ (roughly 7 times as many). While *high-degree twists* (which reduce the cost of $\mathbb{G}_2$ arithmetic relative to $\mathbb{G}_1$ arithmetic) are not available between Edwards curves [CLN10], Edwards curves enjoy one of the fastest group laws for elliptic curves [BL07, BBJ$^+$08], reducing the cost of $\mathbb{G}_1$ operations, which are more numerous for us.[4]

Given an Edwards curve E defined over $\mathbb{F}_q$ such that $E(\mathbb{F}_q)$ (its curve group over $\mathbb{F}_q$) has order divisible by a prime r, one can set $\mathbb{G}_1$ to an order-r subgroup of $E(\mathbb{F}_q)$; then, if E's embedding degree is $k = 6$ (as in our case), $\mathbb{G}_2$ can be set to an order-r subgroup of $E'(\mathbb{F}_{q^3})$, where E' is a *quadratic twist* of E and $E'(\mathbb{F}_{q^3})$ is its group over the larger field $\mathbb{F}_{q^3}$. Thus, group operations in $\mathbb{G}_2$ are indeed more expensive than those in $\mathbb{G}_1$. Finally, $\mathbb{G}_T$ is simply an order-r subgroup of the multiplicative group of $\mathbb{F}_{q^6}$.

Thus, as in [BCG$^+$13], we use the complex-multiplication method [AM93] applied to a curve family of [GMV07] to construct an Edwards curve E, defined over $\mathbb{F}_q$, with embedding degree $k = 6$ such that (i) the size of E's curve group over $\mathbb{F}_q$ is $4r$, and (ii) $r - 1$ is sufficiently *smooth*. The primes r and q have 181 and 183 bits, so E achieves 80-bit security [FST10].

Property (i) implies that $\frac{\log q}{\log r} \approx 1$; this ensures that operations in the order-r groups $\mathbb{G}_1$ and $\mathbb{G}_2$ are not too expensive to implement relative to the group size. Property (ii) ensures that $\mathbb{F}_r^*$ has a primitive root of unity of large-enough power-of-2 order, which enables important optimizations in the key generator (see Section 5.3) and prover (Section 5.4), including allowing the use of radix-2 multiplicative FFTs (which have excellent asymptotic and concrete efficiency).

Next, we discuss optimizations specific to the verifier (Section 5.2), prover (Section 5.3), and generator (Section 5.4).

5.2 An optimized verifier

The verifier's computation consists of two parts:

[4]E.g., Barreto–Naehrig curves [BN06] enjoy sextic twists, but addition (in Jacobian coordinates) costs 11 field additions plus 5 field squarings. For Edwards curves the cost is only 9 field additions plus 1 field squaring [BL13].

- Use the verification key vk and input $\vec{x} \in \mathbb{F}_r^n$ to compute $\mathsf{vk}_{\vec{x}} := \mathsf{vk}_{\mathsf{IC},0} + \sum_{i=1}^{n} x_i \mathsf{vk}_{\mathsf{IC},i}$. (See Step 1 in Figure 5c.)
- Use the verification key vk, value $\mathsf{vk}_{\vec{x}}$, and proof π, to compute 12 pairings and perform the required checks. (See Step 2, Step 3, Step 4 in Figure 5c.)

Quite clearly, the verifier "only" performs $O(n)$ scalar multiplications in $\mathbb{G}_1$, followed by $O(1)$ pairing evaluations. But how fast can the verifier be? After all, fast verification may be critical to some applications (where, e.g., a proof is separately verified by many mutually-untrusting weak devices).

Variable-base multi-scalar multiplication techniques can be used to significantly reduce the number of $\mathbb{G}_1$ operations needed to compute $\mathsf{vk}_{\vec{x}}$. We discuss these, in the context of prover optimizations, in Section 5.3.

But for small n, *the expensive pairing computations dominate*, and it is thus important to reduce their cost as much as possible. By only making "black-box" use of a pairing, the verifier would need to evaluate 12 pairings (Figure 5c), amounting to 12 *Miller loops* plus 12 *final exponentiations*. The straightforward approach is thus to rely on any high-performance pairing library for carrying out these evaluations.

We proceed differently: we first obtain high-performance implementations of *sub-components* of a pairing, and then combine these in a way that is tailored to the verifier protocol. We now summarize how we instantiate this approach.

Short Miller loops. Different curves support different pairings; also, on a given curve, there is more than one pairing. The choice of pairing impacts efficiency. For instance, the "standard" choice of *reduced Tate pairing* [FR94, FMR06] is not the most efficient one on our curve. Instead, we follow Vercauteren [Ver10] and construct an *optimal pairing* on our curve, which requires only $1/2$ the number of *Miller loop iterations* as the reduced Tate pairing. We efficiently compute each Miller loop iteration by adapting, to the optimal pairing case, the formulas of Arène et al. [ALNR11] for fast computation of the reduced Tate pairing on Edwards curves. These efficient computations heavily rely on the fact that $\mathbb{F}_{q^6}$ is suitably represented as a quadratic extension over a cubic one.

Fast final exponentiation. The final step of many pairing computations (including the optimal pairings we use) is a reduction known as *final exponentiation*, which requires exponentiating the Miller loop's output by a certain power. Final exponentiation ensures output uniqueness, so to enable comparisons across pairing evaluations. Since the embedding degree in our case is $k = 6$, the output of the Miller loop is an element $\alpha \in \mathbb{F}_{q^6}$, and the output of the pairing is $\alpha^{\frac{q^6-1}{r}}$.

If performed naively, final exponentiation is *extremely expensive*: it involves exponentiating an element in a very large field ($\mathbb{F}_{q^6}$ has size $\approx 2^{1100}$) by a very large power ($\frac{q^6-1}{r}$ is $\approx 2^{900}$); such an operation in our implementation would cost 1,479.5 µs. Fortunately, the pairing-based cryptography literature contains a host of techniques to select from to make final exponentiation much more efficient.

First, the exponent $\frac{q^6-1}{r}$ factors into $(q^3-1)\cdot(q+1)\cdot\frac{q^2-q+1}{r}$ [SBC+09]. The first two factors are the "easy part": $\beta := \alpha^{(q^3-1)\cdot(q+1)}$ can be computed with 1 inversion, 2 multiplications, and 4 applications of the Frobenius map (which is fast to compute due to the way we represent $\mathbb{F}_{q^6}$).

The "hard part" is computing $\gamma := \beta^{\frac{q^2-q+1}{r}}$. For this, one can write $\frac{q^2-q+1}{r}$ as $\sum_{i=0}^{m} w_i q^i$, for some integer weights $w_1, \ldots, w_m$ of "small norm", and then compute γ via multi-exponentiation as $\prod_{i=1}^{m}(\beta^{q^i})^{w_i}$. There are generic lattice reduction techniques for finding such weights [KKC13], but in our case the curve arises from a curve family so that good solutions can be found by hand [SBC+09]: we can write $\frac{q^2-q+1}{r} = w_0 + w_1 q$ with $|w_0| \approx \sqrt{q}$ and $w_1 = 4$ so that $\gamma := \beta^{w_0+4q}$.

To further simplify computing γ, we use the fact that β lies in a *cyclotomic subgroup* of $\mathbb{F}_{q^6}$, which makes squaring ≈ 3 times faster than for general field elements in $\mathbb{F}_{q^6}$ [GS10].

Overall, our final exponentiation costs $194.9\,\mu\text{s}$, amounting to a $\approx 7.59\times$ improvement over a naive implementation.

Sharing Miller loops and final exponentiations. The verifier computes a product of two pairings in Step 3, and another one in Step 4 (see Figure 5c). A product of pairings can be evaluated faster than evaluating each pairing separately and then multiplying the results [Sol03, Sco05, GS06, Sco07].

Concretely, in a product of m pairings, the Miller loop iterations for evaluating each factor can be carried out in "lock-step" so to share a single *Miller accumulator variable*, using one $\mathbb{F}_{q^6}$ squaring per loop instead of m.

In a similar vein, one can perform a single final exponentiation on the product of the outputs of the m Miller loops, instead of m final exponentiations and then multiplying the results. In fact, since the output of the pairing can be inverted for free (as the element is *unitary* so that inverting equals conjugating [SB04]), the idea of "sharing" final exponentiations extends to a ratio of pairing products. Thus, in the verifier we only need to perform 5, instead of 12, final exponentiations.

Overall, while separately computing 12 optimal pairings costs $13.6\,\text{ms}$, our verifier, by leveraging the aforementioned techniques, performs all the pairing checks in only $8.1\,\text{ms}$, amounting to a $1.68\times$ improvement.

Precomputation by processing the verification key. Of the 12 pairings the verifier needs to evaluate, only one is such that both of its inputs come from the proof π. All other 11 pairing evaluations have one fixed input, coming from the verification key vk (or being a generator of $\mathbb{G}_1$ or $\mathbb{G}_2$).

Whenever one of the two inputs to a pairing is fixed, precomputation techniques apply [GHS02, BLS03, Sco07], especially in the case when the fixed input is the *base point* in Miller's algorithm (which, in our case, holds for 9 out of the 11 pairing evaluations). We thus split the verifier's computation into an *offline phase*, which consists of a one-time precomputation that *only* depends on vk, and a many-time *online phase*, which depends on the precomputed values, input $\vec{x}$, and proof π. More precisely, the result of the offline phase is a *processed verification key* vk^*. While vk^* is longer than vk (but still short), it allows the online phase to be faster. In our implementation, vk^* decreases the total cost of pairing checks from $8.1\,\text{ms}$ to $4.7\,\text{ms}$ (improvement of $1.72\times$); for inputs of n field elements, vk has size $436\,\text{B} + n \cdot 24\,\text{B}$ while vk^* has size $151\,\text{kB} + n \cdot 24\,\text{B}$.

5.3 An optimized prover

The prover's computation consists of two main parts:

- Compute the coefficients $\vec{h}$ of $H(z) := \frac{A(z)B(z)-C(z)}{Z(z)}$. (See Step 4 in Figure 5b.)
- Use the coefficients $\vec{h}$, QAP witness $\vec{s}$, and public key pk to compute π. (See Step 5 in Figure 5b.)

Each part is very costly if naively performed. We summarize how we efficiently perform these computations.

Computing the coefficients of $\boldsymbol{H}$. In general, a computation such as finding the coefficients of H can be efficiently performed by using FFT and Inverse FFT (IFFT) algorithms for fast polynomial evaluation and interpolation. As in [BCG+13], we rely on the fact that $\mathbb{F}_r$ contains a primitive root of unity of large enough order 2^i, use the classical radix-2 multiplicative FFT [CT65] and its inverse. This algorithm, tailored to the special structure of the field $\mathbb{F}_r$, only requires $O(n \log n)$ field operations for degree-n polynomials, and is particularly simple and efficient in practice.

Variable-base multi-scalar multiplication. Computing π in the prover's last step requires solving large instances of *variable*-base multi-scalar multiplication. Given elements $\mathcal{Q}_1, \dots, \mathcal{Q}_n$ all in $\mathbb{G}_1$ or $\mathbb{G}_2$ and scalars $\alpha_1, \dots, \alpha_n \in \mathbb{F}_r$, the prover needs to compute $\langle \vec{\alpha}, \vec{\mathcal{Q}} \rangle := \alpha_1 \mathcal{Q}_1 + \cdots + \alpha_n \mathcal{Q}_n$.

A good choice of multi-scalar multiplication algorithm depends on what distribution we expect the scalars to follow. In the prover, the scalars $\vec{\alpha}$ are one of two types:

- either $\vec{\alpha}$ is the vector $\vec{c} := (1 \circ \vec{s} \circ \delta_1 \circ \delta_2 \circ \delta_3) \in \mathbb{F}_r^{4+m}$, where $\vec{s} := \mathsf{QAPwit}(C, \vec{x}, \vec{a}) \in \mathbb{F}_r^m$ is the QAP witness and $\delta_1, \delta_2, \delta_3 \in \mathbb{F}_r$ are random;
- or $\vec{\alpha}$ is the vector $\vec{h} \in \mathbb{F}_r^{d+1}$ representing the coefficients of the degree-d polynomial H.

The entries in $\vec{h}$ are essentially random. Instead, the entries in $\vec{s}$ depend on the input $(C, \vec{x}, \vec{a})$ to QAPwit; in turn, $(C, \vec{x}, \vec{a})$ depends on our circuit generator (Section 4).

For the case $\vec{\alpha} = \vec{h}$, we use an *in-place* multi-scalar multiplication algorithm [BC89] (and we implement it following the suggestions in [BDL$^+$11]). The algorithm's efficiency relies on a *heuristic*, but avoids the space requirements of non-heuristic algorithms (such as wNAF-based interleaved multiplication [Möl01]) that require storing $2^{w-1}n$ elements for window size w — too much for us because n is very large.

For the case $\vec{\alpha} = \vec{c}$, using the previous algorithm "as is" is inefficient. Indeed, the heuristic algorithm works well when all the scalars have roughly the same bit complexity, but the entries in $\vec{c}$ have very different bit complexity. Nonetheless, we observe that the entries can be "clustered" into just a few groups of scalars with approximately the same bit complexity; thus, we apply the algorithm of [BC89] to each such group.

We stress that, while previous works did leverage multi-scalar multiplication [PGHR13, BCG$^+$13], they did not tailor its use to the specific scalar distributions arising in the protocol.

Proving key in sparse representation. As discussed above, the prover computes $\langle \vec{\alpha}, \vec{\mathcal{Q}} \rangle$ for various choices of scalar vectors $\vec{\alpha}$ (supplied by the prover) and element vectors $\vec{\mathcal{Q}}$ (taken from the proving key pk). Because some of the element vectors contain many zero elements (which do not affect the inner product), the key generator stores these vectors in pk in a *sparse*, rather than dense, form (see Section 5.4). We extend the multi-scalar multiplication algorithms, in the straightforward way, to support element vectors in sparse form, thereby saving time and space for the prover.

5.4 An optimized key generator

The key generator's computation consists of two parts:

- Evaluate each A_i, B_i, C_i at τ.
- Compute the proving key pk and verification key vk. (See Step 3 and Step 4 in Figure 5a.)

Once again, each part is very costly if naively performed. We summarize how we efficiently perform these computations.

Evaluation of A_i, B_i, C_i. As in [BCG$^+$13], the first part can be performed in $O(|C|)$ field operations, by relying on the fact that $\mathbb{F}_r$ has a primitive root of unity of large enough order.

Fixed-base multi-scalar multiplication. The second part is dominated by the computation of pk, due to the large number of entries in pk. This computation requires solving large instances of *fixed*-base multi-scalar multiplication. Namely, given an element $\mathcal{P}$ in $\mathbb{G}_1$ or $\mathbb{G}_2$ and scalars $\alpha_1, \dots, \alpha_n \in \mathbb{F}_r$, the key generator needs to compute $\alpha_1 \mathcal{P}, \dots, \alpha_n \mathcal{P}$.

Again, a good choice of multi-scalar multiplication algorithm depends on what distribution we expect the scalars to follow. In the key generator, the scalars $\vec{\alpha}$ are either zero (see next paragraph) or random evaluations of low-degree polynomials. As in [PGHR13, BCG+13], we find that using fixed-base windowing [BGMW93] is a simple but very efficient solution.

Leveraging sparsity of the QAP. Given an arithmetic circuit C, some of the polynomials in the QAP instance $(\vec{A},\vec{B},\vec{C},Z) = \mathsf{QAPinst}(C)$ are zero. Discussing the details of how the instance map $\mathsf{QAPinst}$ (from Lemma 2.4) works is outside the scope of this paper. Though, very roughly, C's topology (i.e., the graph induced by C's gate connections) determines which polynomials in $\vec{A}$, $\vec{B}$, and $\vec{C}$ are zero.

In practice, many circuits yield a QAP instance for which the density of non-zero polynomials in $\vec{A},\vec{B},\vec{C}$ is well below 100%. For instance, our circuit generator typically outputs a circuit for which the percentage of non-zero polynomials in $\vec{A}$, $\vec{B}$, and $\vec{C}$ is approximately 52%, 15%, 71% respectively.

We reduce the size of the proving key pk (and thus also reduce the space usage of the key generator)by leveraging the sparsity of the QAP instance. Concretely, instead of representing, say, the pk_A component of pk as a dense array of elements, we represent pk_A as a list of position-element pairs corresponding to the non-zero elements. Similarly for the pk'_A, pk_B, pk'_B, pk_C, and pk'_C components of pk. (As for pk_K and pk_H, the remaining components in pk, there is no sparsity to leverage.) For circuits produced by our circuit generator, the size of pk shrinks by 35%.

6 System Evaluation

We evaluated our system on a desktop computer with a 3.40 GHz Intel Core i7-4770 CPU (with Turbo Boost disabled) and 16 GB of RAM. All experiments, except the largest listed in Figure 12 and Figure 13, used a small fraction of the RAM. For the two largest experiments in Figure 13 we increased RAM to 32 GB and added a Crucial M4 solid state disk for swap space.

While our code supports multithreading, for ease of comparison we ran it in single-thread mode.

6.1 Microbenchmarks

Finite field and curve arithmetic. To estimate the costs of our finite-field and curve arithmetic (discussed in Section 5.1), we performed various microbenchmarks, which are reported in Figure 7 and in Figure 8. We remind that for us the primes r and q respectively have 181 and 183 bits (i.e., $\lceil \log_2 r \rceil = 181$).

Pairing computation. The verifier needs to evaluate 12 pairings. However, as discussed in Section 5.2, our implementation does *not* treat the pairing as a black box. Instead, we utilize numerous optimizations, some of which require "sharing" sub-computations of a pairing across multiple pairing evaluations. Thus, we have separately benchmarked the various sub-computations of one optimal pairing evaluation on our curve; see Figure 9. (Benchmarks for the whole verifier are reported in Section 6.3.)

Multi-scalar multiplication. The key generator needs to solve large instances of *fixed*-base multi-scalar multiplication (Section 5.4), while the prover large instances of *variable*-base multi-scalar multiplication (see Section 5.3).

In both cases, we employ a multi-scalar multiplication algorithm. In the first case, we use fixed-base windowing [BGMW93]; in the second case, we use an in-place heuristic algorithm [BC89] (or a variation thereof, depending on the distribution of scalars). See Figure 10 for microbenchmarks comparing the performance, for random points and scalars, of multi-scalar multiplication against naive multiplication in the two cases.

Operation	Time
$\mathbb{F}_r$ add	6.4 ns
$\mathbb{F}_r$ sub	5.7 ns
$\mathbb{F}_r$ mul	26.1 ns
$\mathbb{F}_r$ sqr	21.1 ns
$\mathbb{F}_r$ inv	920.7 ns

Operation	Time
$\mathbb{F}_q$ add	4.2 ns
$\mathbb{F}_q$ sub	5.5 ns
$\mathbb{F}_q$ mul	26.2 ns
$\mathbb{F}_q$ sqr	23.1 ns
$\mathbb{F}_q$ inv	976.1 ns

Operation	Time
$\mathbb{F}_{q^3}$ add	17.3 ns
$\mathbb{F}_{q^3}$ sub	16.8 ns
$\mathbb{F}_{q^3}$ mul	300.8 ns
$\mathbb{F}_{q^3}$ sqr	233.7 ns
$\mathbb{F}_{q^3}$ inv	1,304.7 ns

Operation	Time
$\mathbb{F}_{q^6}$ add	38.2 ns
$\mathbb{F}_{q^6}$ sub	40.2 ns
$\mathbb{F}_{q^6}$ mul	1,052.5 ns
$\mathbb{F}_{q^6}$ sqr	778.4 ns
$\mathbb{F}_{q^6}$ inv	2,568.5 ns

Operation	Time
$\mathbb{F}_{q^6}$ Frobenius map	163.2 ns
$G_{\phi_6(q)}$ squaring	520.7 ns
$\mathbb{F}_{q^6}$ unitary inversion	22.1 ns

Figure 7: **Finite field arithmetic microbenchmarks.** Costs of operations in the fields $\mathbb{F}_r$, $\mathbb{F}_q$, $\mathbb{F}_{q^3}$, and $\mathbb{F}_{q^6}$.

Operation	Time
$\mathbb{G}_1$ add	335.4 ns
$\mathbb{G}_1$ neg	8.1 ns
$\mathbb{G}_1$ dbl	242.1 ns
$\mathbb{G}_1$ compr	1,014.2 ns
$\mathbb{G}_1$ decom	14,957.0 ns

Operation	Time
$\mathbb{G}_2$ add	3,177.8 ns
$\mathbb{G}_2$ neg	20.4 ns
$\mathbb{G}_2$ dbl	2,101.4 ns
$\mathbb{G}_2$ compr	2,904.8 ns
$\mathbb{G}_2$ decom	357,260.4 ns

Figure 8: **Curve arithmetic microbenchmarks.** Costs of operations in the curve groups $\mathbb{G}_1$ and $\mathbb{G}_2$. The labels "compr" and "decom" stand for compression and decompression of curve points.

Operation	Time
Precomputation for $\mathbb{G}_1$ input	1.2 µs
Precomputation for $\mathbb{G}_2$ input	563.8 µs
Miller loop (given precomputation)	283.5 µs
Final exponentiation	106.2 µs

Figure 9: **Pairing computation.** Costs of various sub-computations necessary to evaluate a single optimal pairing on our curve.

Number of exponents	Fixed-base in $\mathbb{G}_1$		Variable-base in $\mathbb{G}_1$	
	Naive	Opt.	Naive	Opt.
10^4	0.8 s	0.7 s	0.8 s	0.1 s
10^5	8.4 s	1.2 s	8.5 s	0.6 s
10^6	84.4 s	6.6 s	84.8 s	6.7 s

Figure 10: **Multi-scalar multiplication** Comparing the costs of naive vs. multi-scalar multiplication in $\mathbb{G}_1$ for fixed and variable bases.

6.2 Performance of our circuit generator

We benchmark our circuit generator for vnTinyRAM. A summary of the construction is in Section 4. For concreteness, when reporting costs in this subsection, we fix $W, K = 16$ (i.e., a machine with 16 registers of 16 bits).

Recall from Definition 3.2 that a circuit generator consists of a circuit generator circ and witness map wit. The circuit generator circ takes as input integers ℓ, n, T, and outputs an $\mathbb{F}_r$-arithmetic circuit

$C\colon \mathbb{F}_r^m \times \mathbb{F}_r^h \to \mathbb{F}_r^l$, for $m := |\ell 2W|_r + |nW|_r$ and some h, l. If the circuit generator has efficiency $f(\cdot)$ then C has $f(\ell, n, T)$ gates.

The circuit C verifies the correct execution of *any* program with $\leq \ell$ instructions, for $\leq T$ steps, on primary inputs of $\leq n$ words. The witness map wit maps a program with accepting inputs into a satisfying assignment for C.

Theorem 4.1 states that our circuit generator has efficiency

$$f(\ell, n, T) := O\big((\ell + n + T) \cdot \log(\ell + n + T)\big) \ .$$

Of course, the above expression only characterizes asymptotic efficiency, so we must discuss the concrete efficiency of our circuit generator. We begin by "uncovering" the constants hidden in the *big-oh* notation, which we estimated by measuring the number of gates in various subcircuits of the generated circuit C. We obtained the following expression:

$$\approx 720T + (200 + 2\log S) \cdot (n + \ell + 2T) + (10 + 2\log S) \cdot S \ ,$$

where S is $(\ell + n + 2T)$ rounded to the next power of 2. Let us go through the above expression so to understand it.

Execution: $720T$. This term arises from T copies of a subcircuit tasked with ensuring validity of instruction execution from one CPU state to the next one in the execution trace. We stress that the interpretation "we incur a $720\times$ overhead" is thoroughly incorrect: T bounds the number of steps of the *abstract* machine vnTinyRAM. Even merely *executing* (not to talk about *verifying*) each step of vnTinyRAM has a *concrete* cost c in terms of gates. So $720T$ must be compared against cT. Of course, we do not know what is the optimal value of c, but, since a CPU state consists of hundreds of bits, c is likely to be in the hundreds, if not even *higher* than 720. (Indeed, 720 is obtained by leveraging nondeterminism wherever possible, which is not available during plain execution!)

Memory: $(200 + 2\log S) \cdot (n + \ell + 2T)$. This term arises from $(n + \ell + 2T)$ copies of a subcircuit tasked with verifying validity of all memory accesses (both instruction fetch and data loads/stores) in the memory-sorted execution trace. The first $(n+\ell)$ copies correspond to the initial stores for initializing memory with program and input; while the remaining $2T$ copies correspond to T instruction fetches and T (potential) data loads/stores.

Routing: $(10 + 2\log S) \cdot S$. This term arises from routing constraints in the Beneš network, which has $2\log S \cdot S$ switches. The rounding effect (S is a power of 2) comes from the fact that a Beneš network routes packets in powers of 2.

Overall, the second and third terms are the ones that we could consider "true overheads", while the first term represents the concrete cost of verifying abstract execution steps.

In Figure 11, we give the gate count in $C := \mathsf{circ}(\ell, n, T)$, including subcounts for the aforementioned three parts,

- for $\ell = 10^4$, $n = 10^2$, and increasing values of T; and
- for $\ell = 10^5$, $n = 10^2$, and increasing values of T.

The gate counts show two things. First, the concrete cost of machine execution almost always accounts for $\approx 50\%$ of the per-cycle count, so *the overhead we incur due to verification is only* $\approx 2\times$. Second, our techniques efficiently handle large programs: for instance, a ten-fold increase in program size, from $\ell = 10^4$ to $\ell = 10^5$, barely impacts the per-cycle gate count, which increases from 1419.5 to only 1441.4 for $T = 2^{20}$.

Comparison with prior circuit generators. The circuit generator in [BCG$^+$13] gives, e.g., per-cycle counts of 10,872 and 100,872 for $\ell = 10^4$ and $\ell = 10^5$ respectively, when $T = 2^{20}$. Thus we achieve $7.6\times$

smaller circuits for 10^4-instruction programs, and $70.0\times$ smaller circuits for 10^5-instructions. Moreover, our superior asymptotics ensure that *our improvement grows, without bound, as ℓ increases*.

A comparison with the circuit generator in [PGHR13] is not well-defined because (i) it supports only programs with no data dependencies, and (ii) its efficiency is not easily specified as a function of the input program. Intuitively, we expect [PGHR13]'s circuit generator to perform better for programs that are "already closer to a circuit", and worse for programs richer in functionality (which comprise most programs in practice). Experimentally, this is what we find.

On the one hand, we wrote a simple C program multiplying two 10×10 matrices of 16-bit integers. The circuit generator in [PGHR13] produces a circuit with 1100 gates, while our circuit generator (when given the corresponding vnTinyRAM assembly) produces a circuit with $\approx 10^7$ gates. Thus, not surprisingly, when invoked on a "circuit-like" program such as matrix multiplication, our circuit generator does not perform well compared to more specific techniques, and for such programs one should use the circuit generator in [PGHR13].

On the other hand, most realistic programs are not "circuit like". For instance, consider a fundamental operation: given an m-integer array, read the integer at a non-constant index $i \in [m]$. Unable to support data-dependent indices, the circuit generator in [PGHR13] forces a programmer to implement array accesses via a *linear scan* (at least without useful information about possible values of i). Established via experiment, the cost, in gates, of such an access in [PGHR13] is $\approx 2m$. In contrast, in vnTinyRAM an array access only costs *one* vnTinyRAM cycle, which usually costs < 1500 gates. Thus, for $m > 10^4$, the cost of linear scan is $> 13\times$ the cost of one vnTinyRAM cycle, and the efficiency gap grows as m increases.

This inefficiency extends beyond array accesses. While vnTinyRAM supports, say, $O(1)$-time operations on hash-tables or $O(\log n)$-time operations on binary search trees or heaps, the circuit generator of [PGHR13] can only emulate these with $O(n)$-time scans. This limitation also precludes efficiency benefits of data structures more complex than arrays.

We find it an intriguing open question whether techniques for handling arbitrary programs such as nondeterministic routing can be effectively combined with more specific techniques [PGHR13] so to yield circuit generators that are efficient both for restricted and rich programs.

$\ell = 10^4$, $n = 10^2$, $T = *$	Num. of gates in $C := \mathsf{circ}(\ell, n, T)$			
	Per cycle	Per cycle divided between		
		exec.	mem.	routing
2^{20}	1,419.5	720.0	487.5	212.0
2^{22}	1,441.6	720.0	493.6	228.0
2^{24}	1,465.2	720.0	501.2	244.0
2^{26}	1,489.0	720.0	509.0	260.0
2^{28}	1,513.0	720.0	517.0	276.0

$\ell = 10^5$, $n = 10^2$, $T = *$	Num. of gates in $C := \mathsf{circ}(\ell, n, T)$			
	Per cycle	Per cycle divided between		
		exec.	mem.	routing
2^{20}	1,441.4	720.0	509.4	212.0
2^{22}	1,447.2	720.0	499.2	228.0
2^{24}	1,466.6	720.0	502.6	244.0
2^{26}	1,489.4	720.0	509.4	260.0
2^{28}	1,513.1	720.0	517.1	276.0

Figure 11: **Number of gates in generated circuit.** Number of gates in the circuit $C := \mathsf{circ}(\ell, n, T)$ as the time bound T increases, for two settings of program size bound ℓ and input bound n. The numbers show that the per-cycle costs are not significantly impacted by ℓ as T increases.

6.3 Performance of our zk-SNARK for circuit satisfiability

We benchmark our implementation of the zk-SNARK for circuit satisfiability from Figure 5. A summary of our efficiency optimizations is given in Section 5.

The key generator takes as input an arithmetic circuit $C \colon \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^l$. Its efficiency mostly depends on the number of gates and wires in C, because these affect the size and degree of the corresponding QAP (see Lemma 2.4) used by the generator. Thus, for the purposes of benchmarking, we ran the generator on a circuit with $\approx 2^i$ gates and wires for $i \in \{10,12,\dots,24\}$ (and fixed values $n = h = l = 100$). In Figure 12 we report the resulting times, together with sub-times corresponding to various sub-computations.

The prover takes as input a proving key pk, input $\vec{x} \in \mathbb{F}_r^n$, and witness $\vec{a} \in \mathbb{F}_r^h$. Its efficiency depends for the most part on the number of gates and wires in C used to generate pk; we thus ran the prover on the proving keys output by the key generator (on the same circuits). In Figure 12 we report the resulting times, together with sub-times for various sub-computations, and the proof size (which is always 230 bytes).

The verifier takes as input a verification key vk generated for C, input $\vec{x} \in \mathbb{F}_r^n$ for C, and proof π. Its efficiency *only* depends on $\vec{x}$. Thus, for the purposes of benchmarking, we ran the verifier on a random input $\vec{x} \in \mathbb{F}_r^n$ with $n = 2^i$ for $i = 0,2,\dots,20$. In Figure 12 we report the resulting times, including the time to preprocess the verification key (which is a *one-time* computation) and to run the verifier.

Discussion. Our measurements demonstrate that our zk-SNARK implementation efficiently scales up to sixteen million gates (for which key generation takes 26 min and proving takes 30 min), on a desktop computer with 32GB RAM. Our verification times are excellent (≈ 0.4 ms per kilobyte of input), and do not suffer from the same space limitations as key generation and proving; indeed, for an n-kilobyte input, the verification key is $\approx 1.02 \cdot n$ kilobytes.

Remark 6.1. Another factor affecting the efficiency of the key generator and prover is the *sparsity* of the QAP instance obtained from the circuit C. (See Section 5.3 and Section 5.4.) Sparsity depends on C's topology, and varies from circuit to circuit. In Figure 12 we reported worst-case numbers in this respect: we only used circuits whose QAP has *no sparsity*. In general, sparser QAP instances make the key generator and prover faster; in particular, sparsity plays a role in our combined system, as the circuits output by our circuit generator induce relatively-sparse QAP instances.

6.4 End-to-end performance

As discussed in Section 3.4 (and Figure 6), we obtain a zk-SNARK for vnTinyRAM by combining our circuit generator for vnTinyRAM (Section 4) and our zk-SNARK for circuit satisfiability (Section 5). We now benchmark the system obtained from this combination.

But *how* should we benchmark this system? Recall from Section 3.2 that a zk-SNARK for vnTinyRAM consists of a triple of algorithms (KeyGen, Prove, Verify). Given bounds ℓ, n, T (for program size, input size, and time),

- the efficiency of KeyGen and Prove depends on ℓ, n, T;
- while the efficiency of Verify depends only on ℓ, n.

Thus, we benchmark the system as follows.

Benchmarking KeyGen. We evaluate KeyGen for various choices of ℓ and T, while keeping n fixed at, say, 100. (Varying ℓ or n affects KeyGen's efficiency in similar ways, so we report times only for a fixed n.) In Figure 13, we report KeyGen's total running time; the subtime for only running the circuit generator circ (the remaining time is spent in the zk-SNARK key generator G); and the sizes of the corresponding proving and verification keys.

Benchmarking Prove. We evaluate Prove for the same choices of ℓ, n, T as for KeyGen. In Figure 13, we report Prove's total running time, and subtime for running the witness map wit (the rest is spent in the zk-SNARK prover P).

	Key generator G						Prover P			
Gate	Total	Subtime for computing			pk	vk	Total	Subtime for computing		π
count	time	QAP at τ	pk	vk	size	size	time	$H(z)$	π	size
2^{10}	0.2 s	2.2 ms	0.2 s	5.4 ms	255.0 kB	2.8 kB	0.2 s	3.6 ms	0.2 s	230 B
2^{12}	0.7 s	8.6 ms	0.6 s	5.1 ms	1.0 MB	2.8 kB	0.6 s	16.3 ms	0.6 s	230 B
2^{14}	2.3 s	34.3 ms	2.2 s	5.1 ms	4.2 MB	2.8 kB	2.2 s	73.0 ms	2.1 s	230 B
2^{16}	7.8 s	137.3 ms	7.7 s	5.1 ms	16.6 MB	2.8 kB	8.3 s	0.3 s	7.9 s	230 B
2^{18}	27.9 s	0.5 s	27.3 s	5.1 ms	66.5 MB	2.8 kB	31.0 s	1.4 s	29.6 s	230 B
2^{20}	100.9 s	2.2 s	98.6 s	5.1 ms	266.1 MB	2.8 kB	119.4 s	6.4 s	113.0 s	230 B
2^{22}	356.9 s	8.8 s	347.5 s	5.1 ms	1.1 GB	2.8 kB	458.6 s	28.0 s	430.6 s	230 B
2^{24}	1,327.7 s	35.1 s	1,290.3 s	5.0 ms	4.3 GB	2.8 kB	1,771.7 s	120.8 s	1,650.9 s	230 B

			Verifier V			
Input size		vk	Preprocess vk	Total time	Subtime for computing	
as field elts.	as bytes	size	(offline)	(online)	$\mathsf{vk}_{\vec{x}}$	pairing checks
2^{0}	23 B	475 B	3.4 ms	4.7 ms	0.1 ms	4.6 ms
2^{2}	90 B	544 B	3.4 ms	4.8 ms	0.2 ms	4.6 ms
2^{4}	360 B	820 B	3.4 ms	4.9 ms	0.3 ms	4.6 ms
2^{6}	1.4 kB	1.9 kB	3.4 ms	5.7 ms	1.1 ms	4.6 ms
2^{8}	5.8 kB	6.3 kB	3.4 ms	8.2 ms	3.6 ms	4.6 ms
2^{10}	23.0 kB	24.0 kB	3.4 ms	16.4 ms	11.8 ms	4.6 ms
2^{12}	92.2 kB	94.7 kB	3.4 ms	45.3 ms	40.7 ms	4.6 ms
2^{14}	368.6 kB	377.3 kB	3.4 ms	151.5 ms	146.9 ms	4.6 ms
2^{16}	1.5 MB	1.5 MB	3.4 ms	555.0 ms	550.3 ms	4.6 ms
2^{18}	5.9 MB	6.0 MB	3.4 ms	2,321.9 ms	2,317.3 ms	4.6 ms
2^{20}	23.6 MB	24.1 MB	3.4 ms	9,556.2 ms	9,551.6 ms	4.6 ms

Figure 12: **Performance of zk-SNARK for arithmetic circuit satisfiability.** Costs of the key generator and prover for various circuit sizes, and of the verifier for various input sizes. In the verifier, the preprocessing of vk is a one-time operation that can be amortized across any number of verifications.

Benchmarking Verify. Since the efficiency of Verify does not depend on T, we evaluate Verify for various choices of ℓ and n. In Figure 13, we report Verify's running time. (All the times assume that the verification key vk has been preprocessed; this one-time operation saves 3.5 ms.)

Remark 6.2. As the program size bound ℓ grows, so does verification time. Yet, if one has to verify many proofs relative to the *same* program $\mathbf{P}$ but different inputs, one can avoid this: verification time can be made *independent* of ℓ, and dependent only on the input size bound n. Namely, given vk and $\mathbf{P}$, one can derive, in time $O(\ell)$, a *program-specific* verification key $\mathsf{vk}_{\mathbf{P}}$. Using $\mathsf{vk}_{\mathbf{P}}$ it is possible, e.g., to reduce verification time to 4.7 ms, 4.8 ms, 5.1 ms, 8.2 ms respectively for $n = 10, 10^2, 10^3, 10^4$ *regardless* of ℓ. The idea behind this technique is evaluating the program-specific part of $\mathsf{vk}_{\vec{x}}$ computation (see Figure 5) ahead of time, so that the online phase would be tasked with just evaluating the input-dependent part of $\mathsf{vk}_{\vec{x}}$ and combining it with the pre-computed result.

6.5 Case study: `memcpy` with just-in-time compilation

The function `memcpy` is a standard C function that works as follows: given as input two array pointers and a length, `memcpy` copies the contents of one array to the other. Of course, with no data dependencies, copying data in a circuit is trivial: you just connect the appropriate wires. However, when the array addresses and their lengths are data dependent, and `memcpy` is invoked as a subroutine in a larger program, an efficient implementation is needed.

Times for KeyGen (and subtimes for circ)			
$n = 100$	$\ell = 10^2$	$\ell = 10^3$	$\ell = 10^4$
$T = 4000$	405.6 s (3.6 s)	417.1 s (3.7 s)	665.5 s (5.9 s)
$T = 8000$	701.3 s (6.2 s)	778.7 s (7.1 s)	902.2 s (8.3 s)
$T = 16000$	1,349.9 s (12.3 s)	1,486.2 s (13.9 s)	1,604.6 s (15.0 s)
$T = 32000$	2,665.6 s (24.2 s)	2,681.2 s (24.3 s)	3,200.5 s (29.0 s)

Sizes of proving and verification keys output by KeyGen						
$n = 100$	$\ell = 10^2$		$\ell = 10^3$		$\ell = 10^4$	
$T = 4000$	1.2 GB	1.1 kB	1.2 GB	4.8 kB	2.0 GB	41.6 kB
$T = 8000$	2.2 GB	1.1 kB	2.5 GB	4.8 kB	2.8 GB	41.6 kB
$T = 16000$	4.3 GB	1.1 kB	4.9 GB	4.8 kB	5.3 GB	41.6 kB
$T = 32000$	8.7 GB	1.1 kB	8.7 GB	4.8 kB	10.2 GB	41.6 kB

Times for Prove (and subtimes for wit)			
$n = 100$	$\ell = 10^2$	$\ell = 10^3$	$\ell = 10^4$
$T = 4000$	154.0 s (5.6 s)	155.0 s (5.8 s)	308.9 s (9.6 s)
$T = 8000$	306.6 s (9.9 s)	314.8 s (11.9 s)	321.9 s (13.8 s)
$T = 16000$	637.5 s (23.3 s)	665.5 s (28.2 s)	726.2 s (30.5 s)
$T = 32000$	1,606.1 s (60.8 s)	1,604.9 s (61.1 s)	1,721.1 s (85.3 s)

Times for Verify			
	$\ell = 10^2$	$\ell = 10^3$	$\ell = 10^4$
$n = 0$	4.7 ms	4.8 ms	5.2 ms
$n = 10$	4.7 ms	4.8 ms	5.2 ms
$n = 10^2$	4.8 ms	4.9 ms	5.2 ms
$n = 10^3$	5.2 ms	5.2 ms	5.5 ms
$n = 10^4$	8.4 ms	8.4 ms	8.7 ms

Figure 13: **Performance of zk-SNARK for vnTinyRAM.** Costs of KeyGen and Prove for various choices of ℓ and T (all with $n = 100$), and of Verify for various choices of ℓ and n. The $T = 32{,}000$ case is slightly slower than interpolation suggests, due to swapping memory to disk.

A naive implementation of `memcpy` iterates, via a loop, over each array position i and copies the i-th value from one array to the other. In vnTinyRAM each such loop iteration costs 6 instructions; 2 of these are to increase the iteration counter and jump back to the start of the loop. Thus, for m-long arrays, copying takes $6m$ instructions (discounting loop initialization). A cost of $6m$ also holds for TinyRAM of [BCG+13].

In contrast, in vnTinyRAM, one can do better. Namely, loop unrolling can be used to avoid paying for the 2 "control" instructions; one can verify that, asymptotically, the optimal number of unrollings *depends* on the array length: it is $\Theta(\sqrt{m})$. This is why *dynamic* loop unrolling is used, and requires the use of a von Neumann architecture. We wrote a 54-instruction vnTinyRAM program for `memcpy` that uses dynamic loop unrolling to achieve an efficiency of $\approx 4m + 11.5\sqrt{m}$ cycles for m-long arrays. For $m \geq 600$, we get $1.25\times$ speed-up over the naive implementation, and $1.4\times$ speed-up for $m \geq 3000$.

7 Other Prior Work

Prior work that is most relevant to us is previous work on zk-SNARKs, which is discussed in Section 1.2. There are also numerous works studying variations or relaxations of the goal we consider; here, we summarize some of them.

Interactive proofs for low-depth circuits. Goldwasser et al. [GKR08] obtained an interactive proof for outsourcing computations of *low-depth circuits*. A set of works [CMT12, TRMP12, Tha13] has optimized and implemented the protocol of [GKR08]. The protocol of [GKR08] can also be reduced to a two-message argument system [KR09, KRR13]. Canetti et al. [CRR12] showed how to extend the techniques in [GKR08] to also handle non-uniform circuits.

Batching arguments. Ishai et al. [IKO07] constructed a *batching argument* for NP, where, to simultaneously verify that ℓ circuits of size S are satisfiable, the verifier runs in time $\max\{S^2, \ell\}$ (i.e., in time (S^2/ℓ) per circuit).

A set of works [SBW11, SMBW12, SVP+12, SBV+13] has improved, optimized, and implemented the batching argument of Ishai et al. [IKO07] for the purpose of outsourcing computation. In particular, by relying on quadratic arithmetic programs of [GGPR13], Setty et al. [SBV+13] have improved the running time of the verifier and prover to $\max\{S, \ell\} \cdot \mathrm{poly}(\lambda)$ and $\tilde{O}(S) \cdot \mathrm{poly}(\lambda)$ respectively.

Vu et al. [VSBW13] provide a system that incorporates both the batching arguments of [SBW11, SMBW12, SVP+12, SBV+13] as well as the interactive proofs of [CMT12, TRMP12, Tha13]. The system decides which of the two approaches is more efficient to use.

Braun et al. [BFR+13] apply batching techniques (as well as zk-SNARKs) to verify MapReduce computations, by relying on various verifiable data structures.

Arguments with competing provers. Canetti et al. [CRR11] use collision-resistant hashes to get a protocol for outsourcing deterministic computations in a model where a verifier interacts with two computationally-bounded provers at least one of which is honest [FK97]. Remarkably, the protocol in [CRR11] works *directly* for random-access machines, and therefore does not require reducing random-access machines to any "lower-level" representation (such as circuits). Canetti et al. implement their protocol for deterministic x86 programs.

Previous circuit generators. Some prior work addressed the problem of translating high-level languages into low-level languages such as circuits. Yet, most prior work only supported limited functionality for high-level languages: [SVP+12, SBV+13] present a circuit generator based on Fairplay [MNPS04, BDNP08], whose SFDL language does not support important primitives and has inefficient support for others; [PGHR13] present a circuit generator for programs without data dependencies.

Some recent works support much more expressive functionality: [BCG+13] relies on nondeterministic routing to support random-access machine computations [BCGT13]; [BFR+13] relies on online memory checking [BEG+91, BCGT13] to support stateful computations.

Other cryptographic tools. *Fully-homomorphic encryption* (FHE) [Gen09] and *probabilistically-checkable proofs* [AS98, ALM+98] are powerful tools that are often used in protocols for outsourcing computations (with integrity or confidentiality guarantees, or both) [Kil92, Mic00, AIK10, GGP10, CKV10, KRR13, GKP+13]. However, such constructions have so far not been explored in practice. Another powerful tool is *secure multi-party computation* [GMW87, BOGW88], but most work in this area does not consider the goal of succinctness.

8 Conclusion & Future Work

We have presented a high-performance implementation of a zk-SNARK for programs written in a relatively expressive model of computation: a simple von Neumann RISC architecture. Proofs are non-interactive and publicly verifiable; moreover, proofs are only a few-hundred bytes long and can typically be verified in just a few milliseconds.

Our system combines a highly-optimized circuit generator for the architecture and a high-performance zk-SNARK for producing and verifying proofs for the resulting arithmetic circuits. As discussed in Section 6, our system significantly improves, both in efficiency and functionality, over previous implementations of succinct non-interactive arguments [PGHR13, BCG$^+$13].

Going forward, we believe that further enriching classes of programs that zk-SNARKs can (efficiently!) support is crucial for catalyzing their use in security applications. Our system efficiently supports a simple von Neumann RISC architecture; we view this as a solid foundation.

More work lies ahead to push the envelope towards richer functionality and better efficiency: from building faster proof systems at the foundations, to applications, such as compilers for architectures like vnTinyRAM. We are eager to see where the journey to succinct verified computation will bring us.

Acknowledgments

We thank Daniel Genkin, Raluca Ada Popa, Ron Rivest, and Nickolai Zeldovich for helpful comments and discussions, and Lior Greenblatt, Shaul Kfir, Michael Riabzev, and Gil Timnat for programming assistance. We also thank Nickolai Zeldovich for the use of his group's compute nodes.

This work was supported by: the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370; the Check Point Institute for Information Security; the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 240258; the Israeli Centers of Research Excellence I-CORE program (center 4/11); the Israeli Ministry of Science and Technology; the Simons Foundation, with a Simons Award for Graduate Students in Theoretical Computer Science; and the Skolkovo Foundation under grant agreement number 6926059.

References

[AIK10] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In *Proceedings of the 37th International Colloquium on Automata, Languages and Programming*, ICALP '10, pages 152–163, 2010.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $o(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, 1983.

[ALM$^+$98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998. Preliminary version in FOCS '92.

[ALNR11] Christophe Arène, Tanja Lange, Michael Naehrig, and Christophe Ritzenthaler. Faster computation of the Tate pairing. *Journal of Number Theory*, 131(5):842–857, 2011.

[AM93] A. O. L. Atkin and F. Morain. Elliptic curves and primality proving. *Mathematics of Computation*, 61:29–68, 1993.

[AS98] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: a new characterization of NP. *Journal of the ACM*, 45(1):70–122, 1998. Preliminary version in FOCS '92.

[BB04] Dan Boneh and Xavier Boyen. Secure identity based encryption without random oracles. In *Proceedings of the 24th Annual International Cryptology Conference*, CRYPTO '04, pages 443–459, 2004.

[BBJ$^+$08] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards curves. In *Proceedings of the 1st International Conference on Cryptology in Africa*, AFRICACRYPT' 08, pages 389–405, 2008.

[BC89] Jurjen Bos and Matthijs Coster. Addition chain heuristics. In *Proceedings of the 9th Annual International Cryptology Conference*, CRYPTO '89, pages 400–407, 1989.

[BCG$^+$13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the 33rd Annual International Cryptology Conference*, CRYPTO '13, pages 90–108, 2013.

[BCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *Proceedings of the 4th Innovations in Theoretical Computer Science Conference*, ITCS '13, pages 401–414, 2013.

[BCI$^+$13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Rafail Ostrovsky, and Omer Paneth. Succinct non-interactive arguments via linear interactive proofs. In *Proceedings of the 10th Theory of Cryptography Conference*, TCC '13, pages 315–333, 2013.

[BDL$^+$11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems*, CHES '11, pages 124–142, 2011.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 257–266, 2008.

[BEG$^+$91] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, FOCS '91, pages 90–99, 1991.

[BFR$^+$13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 341–357, 2013.

[BGDM$^+$10] Jean-Luc Beuchat, Jorge E. González-Díaz, Shigeo Mitsunari, Eiji Okamoto, Francisco Rodríguez-Henríquez, and Tadanori Teruya. High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves. In *Proceedings of the 4th International Conference on Pairing-Based Cryptography*, Pairing '10, pages 21–39, 2010.

[BGMW93] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David B. Wilson. Fast exponentiation with precomputation. In *Proceedings of the 11th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '92, pages 200–207, 1993.

[BL07] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In *Proceedings of the 13th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '07, pages 29–50, 2007.

[BL13] Daniel J. Bernstein and Tanja Lange. Explicit-formulas database. `http://www.hyperelliptic.org/EFD/`, 2013.

[BLS03] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *Proceedings of the 3rd International Conference on Security in Communication Networks*, SCN '02, pages 257–267, 2003.

[BN06] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Proceedings of the 12th International Conference on Selected Areas in Cryptography*, SAC'05, pages 319–331, 2006.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, 1988.

[BS10] Naomi Benger and Michael Scott. Constructing tower extensions of finite fields for implementation of pairing-based cryptography. In *Proceedings of the 3rd International Conference on Arithmetic of Finite Fields*, WAIFI '10, pages 180–195, 2010.

[CKV10] Kai-Min Chung, Yael Kalai, and Salil Vadhan. Improved delegation of computation using fully homomorphic encryption. In *Proceedings of the 30th Annual International Cryptology Conference*, CRYPTO '10, pages 483–501, 2010.

[CLN10] Craig Costello, Tanja Lange, and Michael Naehrig. Faster pairing computations on curves with high-degree twists. In *Proceedings of the 13th International Conference on Practice and Theory in Public Key Cryptography*, PKC '10, pages 224–242, 2010.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the 4th Symposium on Innovations in Theoretical Computer Science*, ITCS '12, pages 90–112, 2012.

[CRR11] Ran Canetti, Ben Riva, and Guy N. Rothblum. Practical delegation of computation using multiple servers. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 445–454, 2011.

[CRR12] Ran Canetti, Ben Riva, and Guy N. Rothblum. Two protocols for delegation of computation. In *Proceedings of the 6th International Conference on Information Theoretic Security*, ICITS 12, pages 37–61, 2012.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[DOhSD06] Augusto Jun Devegili, Colm Ó hÉigeartaigh, Michael Scott, and Ricardo Dahab. Multiplication and squaring on pairing-friendly fields. Cryptology ePrint Archive, Report 2006/471, 2006.

[Edw07] Harold M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(3):393–422, 2007.

[FK97] Uriel Feige and Joe Kilian. Making games short. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, STOC '97, pages 506–516, 1997.

[FMR06] Gerhard Frey, Michael Müller, and Hans-Georg Rück. The Tate pairing and the discrete logarithm applied to elliptic curve cryptosystems. *IEEE Transactions on Information Theory*, 45(5):1717–1719, 2006.

[FR94] Gerhard Frey and Hans-Georg Rück. A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves. *Mathematics of Computation*, 62(206):865–874, 1994.

[FST10] David Freeman, Michael Scott, and Edlyn Teske. A taxonomy of pairing-friendly elliptic curves. *Journal of Cryptology*, 23(2):224–280, 2010.

[Gen04] Rosario Gennaro. Multi-trapdoor commitments and their applications to proofs of knowledge secure under concurrent man-in-the-middle attacks. In *Proceedings of the 24th Annual International Cryptology Conference*, CRYPTO '04, pages 220–236, 2004.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, 2009.

[GES$^+$09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, 2009.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *Proceedings of the 30th Annual International Cryptology Conference*, CRYPTO '10, pages 465–482, 2010.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Proceedings of the 32nd Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '13, pages 626–645, 2013.

[GHS02] Steven D. Galbraith, Keith Harrison, and David Soldera. Implementing the Tate pairing. In *Proceedings of the 5th International Symposium on Algorithmic Number Theory*, ANTS '02, pages 324–337, 2002.

[GKP$^+$13] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the 45th ACM Symposium on the Theory of Computing*, STOC '13, pages 555–564, 2013.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for Muggles. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, STOC '08, pages 113–122, 2008.

[GMV07] Steven D. Galbraith, J. F. Mckee, and P. C. ValençA. Ordinary abelian varieties having small embedding degree. *Finite Fields and Their Applications*, 13(4):800–814, 2007.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, 1987.

[Gro10a] Jens Groth. Short non-interactive zero-knowledge proofs. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '10, pages 341–358, 2010.

[Gro10b] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '10, pages 321–340, 2010.

[GS89] Yuri Gurevich and Saharon Shelah. Nearly linear time. In *Logic at Botik '89, Symposium on Logical Foundations of Computer Science*, pages 108–118, 1989.

[GS06] R. Granger and Nigel Smart. On computing products of pairings. Cryptology ePrint Archive, Report 2006/172, 2006.

[GS10] Robert Granger and Michael Scott. Faster squaring in the cyclotomic subgroup of sixth degree extensions. In *Proceedings of the 13th international conference on Practice and Theory in Public Key Cryptography*, PKC'10, pages 209–223, 2010.

[IKO07] Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short PCPs. In *Proceedings of the Twenty-Second Annual IEEE Conference on Computational Complexity*, CCC '07, pages 278–291, 2007.

[KAK96] Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski, Jr. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, STOC '92, pages 723–732, 1992.

[KKC13] Taechan Kim, Sungwook Kim, and Jung Hee Cheon. On the final exponentiation in Tate pairing computations. *IEEE Transactions on Information Theory*, 59(6):4033–4041, 2013.

[Knu97] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[KR09] Yael Tauman Kalai and Ran Raz. Probabilistically checkable arguments. In *Proceedings of the 29th Annual International Cryptology Conference*, CCC '09, pages 143–159, 2009.

[KRR13] Yael Kalai, Ran Raz, and Ron Rothblum. Delegation for bounded space. In *Proceedings of the 45th ACM Symposium on the Theory of Computing*, STOC '13, pages 565–574, 2013.

[LH00] Chae Hoon Lim and Hyo Sun Hwang. Fast implementation of elliptic curve arithmetic in GF(p^n). In *Proceedings of the Third International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, PKC '00, pages 405–421, 2000.

[Lip12] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *Proceedings of the 9th Theory of Cryptography Conference on Theory of Cryptography*, TCC '12, pages 169–189, 2012.

[Lip13] Helger Lipmaa. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In *Proceedings of the 19th International Conference on the Theory and Application of Cryptology and Information Security*, ASIACRYPT '13, pages 41–60, 2013.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4):1253–1298, 2000. Preliminary version appeared in FOCS '94.

[Mit13] Shigeo Mitsunari. A fast implementation of the optimal ate pairing over BN curve on Intel Haswell processor. Cryptology ePrint Archive, Report 2013/362, 2013.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium*, SSYM '04, pages 20–20, 2004.

[Möl01] Bodo Möller. Algorithms for multi-exponentiation. In *Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography*, SAC '01, pages 165–180, 2001.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[Ofm65] Yuri P. Ofman. A universal automaton. *Transactions of the Moscow Mathematical Society*, 14:200–215, 1965.

[PGHR13] Brian Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, Oakland '13, pages 238–252, 2013.

[Rob91] J. M. Robson. An O(T log T) reduction from RAM computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, May 1991.

[RP06] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953, 2006.

[SB04] Michael Scott and Paulo S. L. M. Barreto. Compressed pairings. In *Proceedings of the 24th Annual International Cryptology Conference*, CRYPTO '04, pages 140–156, 2004.

[SBC$^+$09] Michael Scott, Naomi Benger, Manuel Charlemagne, Luis J. Dominguez Perez, and Ezekiel J. Kachisa. On the final exponentiation for calculating pairings on ordinary elliptic curves. In *Proceedings of the 3rd International Conference Palo Alto on Pairing-Based Cryptography*, Pairing '09, pages 78–88, 2009.

[SBV$^+$13] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J. Blumberg, Bryan Parno, and Michael Walfish. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the 8th EuoroSys Conference*, EuroSys '13, pages 71–84, 2013.

[SBW11] Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Toward practical and unconditional verification of remote computations. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS '11, pages 29–29, 2011.

[Sch78] Claus-Peter Schnorr. Satisfiability is quasilinear complete in NQL. *Journal of the ACM*, 25:136–145, January 1978.

[Sco05] Michael Scott. Computing the Tate pairing. In *Proceedings of the The Cryptographers' Track at the RSA Conference 2005*, CT-RSA '05, pages 293–304, 2005.

[Sco07] Michael Scott. Implementing cryptographic pairings. In *Proceedings of the 1st First International Conference on Pairing-Based Cryptography*, Pairing '07, pages 177–196, 2007.

[SMBW12] Srinath Setty, Michael McPherson, Andrew J. Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *Proceedings of the 2012 Network and Distributed System Security Symposium*, NDSS '12, pages ???–???, 2012.

[Sol03] Jerome A. Solinas. Id-based digital signature algorithms. `http://cacr.uwaterloo.ca/conferences/2003/ecc2003/solinas.pdf`, 2003.

[SVP$^+$12] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the 21st USENIX Security Symposium*, Security '12, pages 253–268, 2012.

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. ArXiv, report 1304.3812, 2013.

[TRMP12] Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. Verifiable computation with massively parallel interactive proofs. *CoRR*, abs/1202.1350, 2012.

[Ver10] Frederik Vercauteren. Optimal pairings. *IEEE Transactions on Information Theory*, 56(1):455–461, 2010.

[VSBW13] Victor Vu, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, Oakland '13, pages 223–237, 2013.