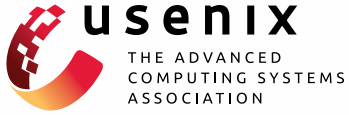

Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture

Eli Ben-Sasson, *Technion—Israel Institute of Technology;* **Alessandro Chiesa,** *Massachusetts Institute of Technology;* **Eran Tromer,** *Tel Aviv University;* **Madars Virza,** *Massachusetts Institute of Technology*

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson

This paper is included in the Proceedings of the 23rd USENIX Security Symposium.

August 20–22, 2014 • San Diego, CA

ISBN 978-1-931971-15-7

Open access to the Proceedings of the 23rd USENIX Security Symposium is sponsored by USENIX

Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture

Eli Ben-Sasson, *Technion*
Alessandro Chiesa, *MIT*
Eran Tromer, *Tel Aviv University*
Madars Virza, *MIT*

Abstract

We build a system that provides succinct non-interactive zero-knowledge proofs (*zk-SNARK*s) for program executions on a von Neumann RISC architecture. 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 prior 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. Moreover, the size of the output circuit depends *additively* (rather than multiplicatively) on program size, allowing verification of larger programs.

The cryptographic proof system improves proving and verification times, by leveraging new algorithms and a pairing 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 also 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.

1 Introduction

1.1 Goal

Consider the setting where a client owns a public input x, a server owns a private input w, and the client wishes to learn $z := F(x,w)$ for a program F known to both parties. For instance, x may be a query, w a confidential database, and F the program that executes the query on the database.

Security. The client is concerned about *integrity* of computation: how can he ascertain that the server reports the correct output z? In contrast, the server is concerned about *confidentiality* of his own input: how can he prevent the client from learning information about w?

Cryptography offers a powerful tool to address these security concerns: *zero-knowledge proofs* [43]. The server, acting as the prover, attempts to convince the client, acting as the verifier, that the following NP statement is true: "there exists w such that $z = F(x,w)$". Indeed:

- The *soundness* property of the proof system guarantees that, if the NP statement is false, the prover cannot convince the verifier (with high probability). Thus, soundness addresses the client's integrity concern.
- The *zero-knowledge* property of the proof system guarantees that, if the NP statement is true, the prover can convince the verifier without leaking any information about w (beyond was is leaked by the output z). Thus, zero knowledge addresses the server's confidentiality.

Moreover, the client sometimes not only seeks soundness but also *proof of knowledge* [43, 11], which guarantees that, whenever he is convinced, not only can he deduce that a witness w exists, but also that the server *knows* one such witness. This stronger property is often necessary to security if F encodes cryptographic computations, and is satisfied by most zero-knowledge proof systems.

Efficiency. Besides the aforementioned security desiderata, many settings also call for *efficiency* desiderata. 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 (e.g., a smartphone).

Thus, it is desirable for the proof to be *non-interactive* [25, 55, 23]: the server just send the claimed output $\tilde{z}$, along with a non-interactive proof string π that attests that $\tilde{z}$ is the correct output. Moreover, it is also desirable for the proof to be *succinct*: π has size $O_\lambda(1)$ and can be verified in time $O_\lambda(|F|+|x|+|z|)$, where $O_\lambda(\cdot)$ is some polynomial in a security parameter λ; in other words, π is very short and easy to verify (i.e., verification time does *not* depend on $|w|$, nor F's running time).

zk-SNARKs. A proof system achieving the above security and efficiency desiderata is called a (publicly-verifiable) *zero-knowledge Succinct Non-interactive ARgument of Knowledge* (zk-SNARK). zk-SNARK constructions can be applied to a wide range of security applications, provided these constructions deliver good enough *efficiency*, and support rich enough *functionality* (i.e., the class of programs F that is supported).

Remark 1.1. In the zero-knowledge setting above, the client does not have the server's input, and so cannot conduct the computation on his own. Hence, it is *not meaningful* to compare "efficiency of outsourced computation at the server" and "efficiency of native execution at the client", because the latter was never an option. Non-interactive zero-knowledge proofs (and zk-SNARKs) are useful regardless of *cross-over points*.

Our goal in this paper is to construct

a zk-SNARK implementation supporting executions on a universal von Neumann RISC machine.

1.2 Prior work

zk-SNARKs. Many works have obtained zk-SNARK constructions [45, 51, 38, 22, 56, 16, 52, 27]. Three of these [56, 16, 27] provide implementations, and thus we briefly recall them. Parno et al. [56] present two main contributions.

- A zk-SNARK, with essentially-optimal asymptotics, for arithmetic circuit satisfiability, based on *quadratic arithmetic programs* (QAPs) [38]. They accompany their construction with an implementation.
- 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. [16] present three main contributions.

- Also a QAP-based zk-SNARK with essentially-optimal asymptotics for arithmetic circuit satisfiability, and a corresponding implementation. Their construction follows the linear-interactive proofs of [22].
- 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.

Finally, Braun et al. [27] re-implemented the protocol of [56] and combined it with a circuit generator that incorporates memory-checking techniques [24] to support random-access memory [14].

Outsourcing computation to powerful servers. Numerous works [63, 65, 66, 64, 32, 68, 71, 67, 27] seek to verifiably outsource computation to untrusted powerful servers, e.g., in order to make use of cheaper cycles or storage. (See Appendix A for a summary.) We stress that verifiable outsourcing of computations *is not our goal*. Rather, as mentioned, we study functionality and efficiency aspects of *non-interactive zero-knowledge proofs*, which are useful even when applied to relatively-small computations, and even with high overheads.

Compared to most protocols to outsource computations, known zk-SNARKs use "heavyweight" techniques, such as *probabilistically-checkable proofs* [6] and expensive pairing-based cryptography. The optimal choice of protocol, and whether it actually pays off compared to local *native execution*, are complex, computation-dependent questions [71], and we leave to future work the question of whether zk-SNARKs are useful for the goal of outsourcing computations.

1.3 Limitations of prior work

Recent work has made tremendous progress in taking zk-SNARKs from asymptotic theory into concrete implementations. Yet, known implementations suffer from several limitations.

Per-program key generation. As in any non-interactive zero-knowledge proof, a zk-SNARK requires a one-time trusted setup of public parameters: a *key generator* samples a proving key (used to generate proofs) and a verification key (used to check proofs). However, current zk-SNARK implementations [56, 16] require the setup phase to depend on the program F, which is *hard-coded* in the keys. Key generation is costly (quasilinear in F's runtime) and is thus difficult to amortize if conducted anew for each program. More importantly, per-program key generation requires, *for each new choice of program*, a trusted party's help.

Limited support for high-level languages. Known circuit generators have limited functionality or efficiency: (i) [56]'s circuit generator only supports programs without data dependencies, since memory accesses and loop iteration bounds cannot depend on a program's input; (ii) [27]'s circuit generator allows data-dependent memory accesses, but each such access requires expensive hashing to verify Merkle-tree authentication paths; (iii) [16]'s circuit generator supports arbitrary programs but its circuit size scales inefficiently with program size (namely, it has size $\Omega(\ell T)$ for ℓ-instruction T-step TinyRAM programs). Moreover, while there are techniques that mitigate some of the above limitations [72], these only apply in special cases, and not do address general data dependencies, a common occurrence in many programs.

Generic sub-algorithms. The aforementioned zk-SNARKs use several sub-algorithms, and in particular elliptic curves and pairings. Protocol-specific optimizations are a key ingredient in fast implementations of

pairing-based protocols [59], yet prior implementations only utilize off-the-shelf cryptographic libraries, and miss key optimization opportunities.

1.4 Results

We present two main contributions: a new circuit generator and a new zk-SNARK for circuits. These can be used independently, or combined to obtain an overall system.

1.4.1 A new circuit generator

We design and build a new circuit generator that incorporates the following two main improvements.

(1) 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. Instead, all prior circuit generators [66, 64, 56, 16, 27] hardcoded the program in the circuit. Combined with a zk-SNARK for circuits (or any NP proof system for circuits), we achieve a notable conceptual advance: *once-and-for-all key generation* that allows verifying all programs up to a given size. This removes major issues in all prior systems: expensive per-program key generation, and the thorny issue of conducting it anew in a trusted way for every program.

Our circuit generator supports a universal machine that, like modern computers, follows the *von Neumann paradigm* (program and data lie in the same read/write address space). Concretely, it supports a von Neumann RISC architecture called vnTinyRAM, a modification of TinyRAM [17]. Thus, we also support programs leveraging techniques such as *just-in-time compilation* or *self-modifying code* [36, 58].

To compile C programs to the vnTinyRAM machine language, we ported the GCC compiler to this architecture, building on the work of [16].

See Figure 1 for a functionality comparison with prior circuit generators (for details, see [27, §2]).

Supported functionality	[66, 64, 56]	[16]	[27]	**this work**
side-effect free comp.	✓	✓	✓	✓
data-dep. mem. accesses	×	✓	✓	✓
data-dep. contr. flow	×	✓	×	✓
self-modifying code	×	×	×	✓
universality	×	×	×	✓

Figure 1: Functionality comparison among circuit generators.

(2) Our circuit generator efficiently handles *larger* arbitrary programs: the size of the generated circuit $C_{\ell,n,T}$ is $O\big((\ell+n+T)\cdot\log(\ell+n+T)\big)$ gates. Thus, the dependence on program size is *additive*, instead of multiplicative as in [16], where the generated (non-universal) circuit has size $\Theta\big((n+T)\cdot(\log(n+T)+\ell)\big)$. As Figure 2 shows, our efficiency improvement compared to [16] is not merely asymptotic but yields sizable concrete savings: as program size ℓ increases, our amortized per-cycle gate count is essentially unchanged, while that of [16] grows without bound, becoming orders of magnitudes more expensive.

$n = 10^2$	$\lvert C_{\ell,n,T}\rvert / T$		improvement
$T = 2^{20}$	[16]	**this work**	
$\ell = 10^3$	1,872	1,368	1.4×
$\ell = 10^4$	10,872	1,371	7.9×
$\ell = 10^5$	100,872	1,400	72.1×
$\ell = 10^6$	1,000,872	1,694	590.8×

Figure 2: Per-cycle gate count improvements over [16].

An efficiency comparison with other non-universal circuit generators [66, 64, 56, 27] is not well-defined. First, they support more restricted classes of programs, so a programmer must "write around" the limited functionality. Second, their efficiency is not easily specified, since the output circuit is ad hoc for the given program, and the only way to know its size is to actually run the circuit generator. We expect, and find, that such circuit generators perform better than ours for programs that are already "close to a circuit", and worse for programs rich in data-dependent memory accesses and control flow.

1.4.2 A new zk-SNARK for circuits

Our third contribution is a high-performance implementation of a zk-SNARK for arithmetic circuits.

(3) We improve upon and implement the protocol of Parno et al. [56]. Unlike previous zk-SNARK implementations [56, 16, 27], we do not use off-the-shelf cryptographic libraries. Rather, we create a tailored implementation of the requisite components: the underlying finite-field arithmetic, elliptic-curve group arithmetic, pairing-based checks, and so on.

To facilitate comparison with prior work, we instantiate our techniques for two specific algebraic setups: we provide an instantiation based on Edwards curves [33] at 80 bits of security (as in [16]), and an instantiation based on Barreto–Naehrig curves [9] at 128 bits of security (as in [56, 27]).

On our reference platform (a typical desktop), proof verification is fast: at 80-bit security, for an n-byte input to the circuit, verification takes $4.7 + 0.0004 \cdot n$ milliseconds, *regardless of circuit size*; at 128-bit security, it takes $4.8 + 0.0005 \cdot n$. The constant term dominates for small inputs, and corresponds to the verifier's pairing-based checks; in both cases, it 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 for a custom implementation of the underlying math.

Key generation and proof generation entail a per-gate cost. For example, for a circuit with 16 million gates: at 80 bits of security, key generation takes 81 μs per gate and proving takes 109 μs per gate; at 128 bits of security, these per-gate costs mildly increase to 100 μs and 144 μs.

As in previous zk-SNARK implementations, proofs have constant size (independent of the circuit or input size); for us, they are 230 bytes at 80 bits of security, and 288 bytes at 128 bits of security.

Compared to previous implementations of zk-SNARKs for circuits [56, 16, 27], our implementation improves both proving and verification times, e.g., see Figure 3.

	80 bits of security			128 bits of security		
	[16]	**this**	impr.	[56]	**this**	impr.
Key gen.	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	4.9 ms	13.5×	9.2 ms	5.1 ms	1.8×
Proof	322 B	230 B	1.4×	288 B	288 B	(same)

Figure 3: Comparison with prior zk-SNARKs for a 1-million-gate arithmetic circuit and a 1000-bit input, running on our benchmarking machine, using software provided by the respective authors. Since [27] is a re-implementation of [56], we only include the latter's performance. ($N = 5$ and std $< 2\%$)

1.4.3 Two components: independent or combined

Our new circuit generator and our new zk-SNARK for circuits can be used independently. For instance, the circuit generator can (up to interface matching) replace the circuit generators in [66, 64, 56, 16, 27], thus granting these systems universality. Similarly, our zk-SNARK for circuits can replace the underlying zk-SNARKs in [56, 16, 27], or be used directly in applications where a suitable circuit is already specified.

Combining these two components, we obtain a full system: a *zk-SNARK for proving/verifying correctness of* vnTinyRAM *computations*; see Figure 4 and Figure 5 for diagrams of this system. We evaluated this overall system for programs with up to 10,000 instructions, running for up to 32,000 steps. Verification time is, again, only few milliseconds, independent of the running time of the vnTinyRAM program, even when program size and input size are kilobytes. Proofs, as mentioned, have a small constant size. Key generation and proof generation entail a per-cycle cost, with a dependence on program size that "tapers off" as computation length increases. For instance, at 128-bit security and vnTinyRAM with a word size of 32 bits, key generation takes 210 ms per cycle and proving takes 100 ms per cycle, for 8K-instruction programs.

JIT case study: efficient `memcpy`. Besides evaluating individual components, we give an example demonstrating the rich functionality supported by the integrated system. We wrote a vnTinyRAM implementation of `memcpy` that leverages *just-in-time compilation* (in particular, *dynamic loop unrolling*) to require fewer cycles. (See Section B.)

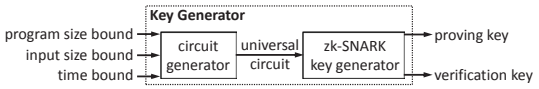

Figure 4: **Offline phase (once).** The key generator outputs a proving key and verification key, for proving and verifying correctness of any program execution meeting the given bounds.

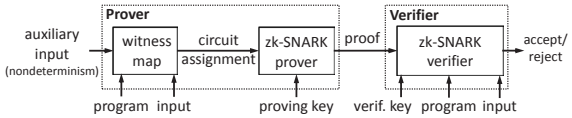

Figure 5: **Online phase (any number of times).** The prover sends a short and easy-to-verify proof to a verifier. This can be repeated any number of times, each time for a different program and input.

2 Preliminaries

$\mathbb{F}[z]$ denotes the ring of univariate polynomials over $\mathbb{F}$, and by $\mathbb{F}^{\leq d}[z]$ the subring of polynomials of degree $\leq d$. Concatenation of vectors/scalars is denoted by $\circ$.

2.1 Arithmetic circuits

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 $\mathbb{F}$. The circuits we consider only have *bilinear gates*.[1]

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 $\mathscr{R}_C = \{(\vec{x},\vec{a}) \in \mathbb{F}^n \times \mathbb{F}^h : C(\vec{x},\vec{a}) = 0^l\}$ and language $\mathscr{L}_C = \{\vec{x} \in \mathbb{F}^n : \exists \vec{a} \in \mathbb{F}^h,\ C(\vec{x},\vec{a}) = 0^l\}$.*

All the arithmetic circuits we consider are over prime fields $\mathbb{F}_p$. 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 $s \in \{0,1\}^m$, we use $[\![s]\!]_p^m$ to denote the vector $\vec{x} \in \mathbb{F}_p^{|m|_p}$, where $|m|_p := \lceil m / \lfloor \log p \rfloor \rceil$, such that the binary representation of $x_i \in \mathbb{F}_p$ is the i-th block of $\lceil \log p \rceil$ bits in s (padded with 0's if needed). We extend the notation $[\![s]\!]_p^m$ to binary strings $s \in \{0,1\}^n$ with $n < m$ bits via padding: $[\![s]\!]_p^m := [\![s0^{m-n}]\!]_p^m$.

2.2 Quadratic arithmetic programs

Our zk-SNARK leverages *quadratic arithmetic programs* (QAPs), introduced by Gennaro et al. [38].

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 $\vec{A},\vec{B},\vec{C}$ are three vectors, each of $m+1$ polynomials in $\mathbb{F}^{\leq d-1}[z]$, and $Z \in \mathbb{F}[z]$ has degree exactly d.*

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

Like a circuit, a QAP induces a satisfaction problem:

Definition 2.3. *The* **satisfaction problem** *of a size-m QAP* $(\vec{A},\vec{B},\vec{C},Z)$ *is the relation* $\mathscr{R}_{(\vec{A},\vec{B},\vec{C},Z)}$ *of pairs* $(\vec{x},\vec{s})$ *such that (i)* $\vec{x}\in\mathbb{F}^n$, $\vec{s}\in\mathbb{F}^m$, *and* $n\leq m$; *(ii)* $x_i=s_i$ *for* $i\in[n]$ *(i.e.,* $\vec{s}$ *extends* $\vec{x}$*); and (iii) the polynomial* $Z(z)$ *divides the following one:*

$$(A_0(z)+\textstyle\sum_{i=1}^m s_iA_i(z))\cdot(B_0(z)+\sum_{i=1}^m s_iB_i(z)) \\ -(C_0(z)+\textstyle\sum_{i=1}^m s_iC_i(z))\ .$$

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

Gennaro et al. [38] showed that circuit satisfiability can be efficiently reduced to QAP satisfiability (which can then be proved and verified using zk-SNARKs):

Lemma 2.4. *There exist two polynomial-time algorithms* $\mathsf{QAPinst},\mathsf{QAPwit}$ *that work as follows. For any circuit* $C\colon\mathbb{F}^n\times\mathbb{F}^h\to\mathbb{F}^l$ *with a wires and b gates,* $(\vec{A},\vec{B},\vec{C},Z):=\mathsf{QAPinst}(C)$ *is a QAP of size m and degree d over* $\mathbb{F}$ *that satisfies the following three properties.*

- EFFICIENCY. *It holds that* $m=a$ *and* $d=b+l+1$.
- COMPLETENESS. *For any* $(\vec{x},\vec{a})\in\mathscr{R}_C$, *it holds that* $(\vec{x},\vec{s})\in\mathscr{R}_{(\vec{A},\vec{B},\vec{C},Z)}$, *where* $\vec{s}:=\mathsf{QAPwit}(C,\vec{x},\vec{a})$.
- PROOF OF KNOWLEDGE. *For any* $(\vec{x},\vec{s})\in\mathscr{R}_{(\vec{A},\vec{B},\vec{C},Z)}$, *it holds that* $(\vec{x},\vec{a})\in\mathscr{R}_C$, *where* $\vec{a}$ *is a prefix of* $\vec{s}$.
- NON-DEGENERACY. *The polynomials* $A_0,\ldots,A_n$ *are nonzero and distinct.*

2.3 Pairings

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 $\mathscr{P},\mathscr{Q}$. We write $\mathbb{G}_1$ and $\mathbb{G}_2$ in additive notation. Let $\mathscr{P}_1$ be a generator of $\mathbb{G}_1$, i.e., $\mathbb{G}_1=\{\alpha\mathscr{P}_1\}_{\alpha\in\mathbb{F}_r}$ (α is also viewed as an integer, hence $\alpha\mathscr{P}_1$ is well-defined); let $\mathscr{P}_2$ 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: (i) *bilinearity*: for every nonzero elements $\alpha,\beta\in\mathbb{F}_r$, it holds that $e(\alpha\mathscr{P}_1,\beta\mathscr{P}_2)=e(\mathscr{P}_1,\mathscr{P}_2)^{\alpha\beta}$; (ii) *non-degeneracy*: $e(\mathscr{P}_1,\mathscr{P}_2)$ is not the identity in $\mathbb{G}_T$.

2.4 zk-SNARKs for arithmetic circuits

A (preprocessing) **zk-SNARK** for $\mathbb{F}$-arithmetic circuit satisfiability (see, e.g., [22]) is a triple of polynomial-time algorithms (G,P,V), called *key generator*, *prover*, and *verifier*. The key generator G, given a security parameter λ and an $\mathbb{F}$-arithmetic circuit $C\colon\mathbb{F}^n\times\mathbb{F}^h\to\mathbb{F}^l$, samples a *proving key* pk and a *verification key* vk; these are the proof system's public parameters, which need to be generated only once per circuit. After that, anyone can use pk to generate non-interactive proofs for the language $\mathscr{L}_C$, and anyone can use the vk to check these proofs. Namely, given pk and any $(\vec{x},\vec{a})\in\mathscr{R}_C$, the honest prover $P(\mathsf{pk},\vec{x},\vec{a})$ produces a proof π attesting that $\vec{x}\in\mathscr{L}_C$; the verifier $V(\mathsf{vk},\vec{x},\pi)$ checks that π is a valid proof for $\vec{x}\in\mathscr{L}_C$. A proof π is both a proof of knowledge, and a (statistical) zero-knowledge proof. The succinctness property requires that π has length $O_\lambda(1)$ and V runs in time $O_\lambda(|\vec{x}|)$, where O_λ hides a (fixed) polynomial in λ.

Constructions. Several zk-SNARK constructions are known [45, 51, 38, 22, 56, 16, 52]. The most efficient ones are based on *quadratic span programs* (QSPs) [38, 52] or *quadratic arithmetic programs* (QAPs) [38, 22, 56, 16]. We focused on QAP-based constructions, because QAPs allow for tighter reductions from *arithmetic* circuits (see Lemma 2.4). Concretely, we build on the QAP-based zk-SNARK protocol of Parno et al. [56] (see Section 4).

Remark 2.5 (full succinctness)**.** The key generator G takes C as input, and so its complexity is linear in $|C|$. One could require G to *not* take C as input, and have its output keys work for *all* (polynomial-size) circuits C; then, G's running time would be independent of C. A zk-SNARK satisfying this stronger property is *fully succinct*. Theoretical constructions of such zk-SNARKs are known, based on various cryptographic assumptions [54, 69, 21]. Despite achieving essentially-optimal asymptotics [6, 18, 15, 14, 21] no implementations of them have been reported to date.

2.5 A von Neumann RISC architecture

Ben-Sasson et al. [16] introduced TinyRAM, a Harvard RISC architecture with word-addressable memory. We modify TinyRAM to obtain vnTinyRAM, which differs from it in two main ways. First, 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. Second, vnTinyRAM has byte-addressable memory, along with instructions to load/store bytes or words.[2]

Besides the above main differences, vnTinyRAM is very similar to TinyRAM. Namely, it is parametrized by the *word size*, denoted W, and the *number of registers*, denoted K. The *CPU state* of the machine consists of (i) a W-bit *program counter*; (ii) K general-purpose W-bit *registers*; (iii) a 1-bit *condition flag*. The full state of the machine also includes *memory*, which is a linear array of 2^W bytes, and two *tapes*, each with a string of W-bit words, and read-only in one direction. One tape is for a *primary input* $\mathbb{x}$ and the other for an *auxiliary input* $\mathbb{w}$ (treated as nondeterministic, untrusted advice).

[2] Byte-addressing is common in programs performing array or string operations (and is a deeply-ingrained assumption in the GCC and LLVM compilers), while word-addressing in programs performing arithmetic.

In memory, an instruction is represented as a double word (one word for an immediate, and another for opcode, etc.). Thus, a *program* $\mathbb{P}$ is a list of address/double-word pairs specifying the initial contents of memory; all other memory locations assume the initial value of 0.

We define the language of accepting computations:

Definition 2.6. *Fix bounds ℓ, n, T. The language $\mathscr{L}_{\ell,n,T}$ consists of pairs $(\mathbb{P}, \mathbb{x})$ such that: (i) $\mathbb{P}$ is a program with $\leq \ell$ instructions, (ii) $\mathbb{x}$ is a primary input with $\leq n$ words, (iii) there exists an auxiliary input $\mathbb{w}$ s.t. $\mathbb{P}(\mathbb{x}, \mathbb{w})$ accepts in $\leq T$ steps. We denote by $\mathscr{R}_{\ell,n,T}$ the relation corresponding to $\mathscr{L}_{\ell,n,T}$.*

3 Our circuit generator

A circuit generator translates the correctness of suitably-bounded program executions into circuit satisfiability: 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. More precisely, using the notations $[\![s]\!]_p$ (for packing the binary string s into field elements) and $|s|_p$ (for computing the number of field elements required to pack s) introduced in Section 2.1, we define a (universal) circuit generator for vnTinyRAM as follows.

Definition 3.1. *A* **(universal) circuit generator** *of efficiency $f(\cdot)$ over a prime field $\mathbb{F}_p$ is a polynomial-time algorithm* circ*, together with an efficient* witness map wit*, working as follows. For any program size bound ℓ, time bound T, and primary-input size bound n, $C := \mathsf{circ}(\ell, n, T)$ is an $\mathbb{F}_p$-arithmetic circuit $C\colon \mathbb{F}_p^m \times \mathbb{F}_p^h \to \mathbb{F}_p^l$, for $m := |\ell 2W|_p + |nW|_p$ and some h, l, where W is the word size (cf. Section 2.5).*

- EFFICIENCY. *The circuit C has $f(\ell, n, T)$ gates.*
- COMPLETENESS. *Given any program $\mathbb{P}$, primary input $\mathbb{x}$, and witness $\mathbb{w}$ such that $\big((\mathbb{P}, \mathbb{x}), \mathbb{w}\big) \in \mathscr{R}_{\ell,n,T}$, it holds that $(\vec{x}, \vec{a}) \in \mathscr{R}_C$, where $\vec{x} := [\![\mathbb{P}]\!]_p^{\ell 2W} \circ [\![\mathbb{x}]\!]_p^{nW}$ and $\vec{a} := \mathsf{wit}(\ell, n, T, \mathbb{P}, \mathbb{x}, \mathbb{w})$.*
- PROOF OF KNOWLEDGE. *There is a polynomial-time algorithm such that, given any $(\vec{x}, \vec{a}) \in \mathscr{R}_C$, outputs a witness $\mathbb{w}$ for $(\mathbb{P}, \mathbb{x}) \in \mathscr{L}_{\ell,n,T}$.*

The circuit C output by circ is *universal* because it does not depend on the program $\mathbb{P}$ or primary input $\mathbb{x}$, but only on their respective size bounds ℓ and n (as well as the time bound T). When combined with any proof system for circuit satisfiability (e.g., our zk-SNARK), this fact enables the generation of the proof systems' parameters to be universal as well. Namely, it is possible to generate keys for all bound choices (e.g., in powers of 2) up to some constant, *once and for all*; afterwards, one can pick the keys corresponding to bounds fitting a given computation. This avoids expensive per-program key generation and, more importantly, the need for a trusted party to conduct key generation anew for every program.

We construct a universal circuit generator with the following efficiency:

Theorem 3.2. *There is a circuit generator of efficiency $f(\ell, n, T) = O\big((\ell + n + T) \cdot \log(\ell + n + T)\big)$ over any prime field $\mathbb{F}_p$ of size $p > 2^{2W}$, where W is the word size (cf. Section 2.5).*

(In our case, the condition $p > 2^{2W}$ is always fulfilled.)

3.1 Past techniques

Most of the difficulties that arise when designing a circuit generator have to do with *data dependencies*. A circuit's topology does not depend on its inputs but, in contrast, program flow and memory accesses depend on the choice of program and the program's inputs. Thus, a circuit tasked with verifying program executions must be "ready" to support a multitude of program flows and memory accesses, despite the fact that its topology has already been fixed. Various techniques have been applied to the design of circuit generators.

Program analysis. In the extreme, if both the program $\mathbb{P}$ and its inputs $(\mathbb{x}, \mathbb{w})$ are known in advance, designing a circuit generator is simple: construct a circuit that evaluates $\mathbb{P}$ on $(\mathbb{x}, \mathbb{w})$ by preparing the circuit's topology to match the pre-determined program flow and memory accesses. But now suppose that only $\mathbb{P}$ is known in advance, but not its inputs $(\mathbb{x}, \mathbb{w})$. In this case, by analyzing $\mathbb{P}$ piece by piece (e.g., separately examine the various loops, branches, and so on), one could try to design a circuit $C_{\mathbb{P}}$ that can handle different choices of inputs. Most prior circuit generators [66, 64, 56, 27] take this approach.

However, this approach suffers from several limitations. First, the class of supported programs $\mathbb{P}$ is not rich, because support for data dependencies is limited. E.g., [56] requires array accesses and loop iteration bounds to be compile-time constants; also, while [27] supports data-dependent memory accesses, most program flow is also restricted to be known (or bounded) at compile-time; mitigations are possible, but only in special cases [72]. Second, and more importantly, this approach does not seem to allow for designing universal circuit generators, because the program $\mathbb{P}$ is *not known in advance* and thus there is no program to analyze.

Multiplex every access. Computers are universal random-access machines (RAMs), so one approach of designing a universal circuit is to mimic a computer's execution, building a layered circuit 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. While this approach supports arbitrary program flow, memory accesses are inefficiently supported; indeed, if memory has S addresses, the resulting circuit is huge: it has size $\Omega(TS)$.

Nondeterministic routing. Ben-Sasson et al. [14] suggested using *nondeterministic routing* on a Beneš network to support memory accesses efficiently; Our circuit generator builds on the techniques of [14, 16], so we briefly review the main idea behind nondeterministic routing.

Following [14], Ben-Sasson et al. [16] introduced a simple computer architecture, called TinyRAM, and constructed a routing-based circuit generator for TinyRAM. They define the following notions. A *CPU state*, denoted S, is the CPU's contents (e.g., program counter, registers, flags) at a given time step. An *execution trace* for a program $\mathbb{P}$, time bound T, and primary input $\mathtt{x}$ is a sequence $\mathsf{tr} = (S_1, \ldots, S_T)$ of CPU states. An execution trace tr is *valid* if there is an auxiliary input $\mathtt{w}$ such that the execution trace induced by $\mathbb{P}$ running on inputs $(\mathtt{x}, \mathtt{w})$ is tr.

We seek an arithmetic circuit C for verifying that tr is valid. We break this down by splitting validity into three sub-properties: (i) *validity of instruction fetch* (for each time step, the correct instruction is fetched); (ii) *validity of instruction execution* (for each time step, the fetched instruction is correctly executed); and (iii) *validity of memory accesses* (each 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 $C_{\mathbb{P}}$ so that, for any two CPU states S and S', $C_{\mathbb{P}}(S, S', g)$ is satisfied for some "guess" g if and only if S' can be reached from S (by fetching from $\mathbb{P}$ the instruction indicated by the program counter in S and then executing it), for *some* state of memory. Then, properties (i) and (ii) hold if $C_{\mathbb{P}}(S_i, S_{i+1}, \cdot)$ is satisfiable for $i = 1, \ldots, T-1$. Thus, C contains $T-1$ copies of $C_{\mathbb{P}}$, 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 equals to the sorting of tr by accessed memory addresses (breaking ties via timestamps), and 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 consistency of "load-after-load", "load-after-store", and so on.) However, C merely gets some auxiliary input tr^*, which *purports* to be $\mathsf{MemSort}(\mathsf{tr})$. So C works as follows: (a) C has $T-1$ copies of C_{mem}, each wired to a pair of adjacent states in tr^*; (b) C separately 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 (i.e., additional values in the auxiliary input), and the routing network merely *verifies* that the switch settings induce a permutation; this allows for a very tight reduction. (Known constructions that *compute* the correct permutation hide large constants in big-oh notation [1].)

Past inefficiencies. After filling in additional details, the construction of [16] reviewed 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 all of the ℓ instructions in $\mathbb{P}$ are *hardcoded* into each of the $T-1$ copies of $C_{\mathbb{P}}$. Thus, besides being non-universal, the circuit scales inefficiently as ℓ grows (e.g., for $\ell = 10^4$, $C_{\mathbb{P}}$'s size is already dominated by $\mathbb{P}$'s size).

3.2 Our construction

In comparison to [16], our circuit generator is universal and, moreover, its size only grows with $\ell + T$ (additive dependence on program size) instead of with $\ell \cdot T$ (multiplicative dependence). As our evaluation demonstrates (see Section 5.1), the size improvement actually translates into significant savings in practice.

Instead of hardcoding the program $\mathbb{P}$ into each copy of the circuit $C_{\mathbb{P}}$, we follow the von Neumann paradigm, where the program $\mathbb{P}$ lies in the same read/write memory space as data. We ensure that $\mathbb{P}$ is loaded into the initial state of memory, using a dedicated circuit; we then verify instruction fetch via the *same* routing network that is used for checking data loads/stores. While the idea is intuitive, realizing it involves numerous technical difficulties, some of which are described below.

Routing instructions and data. We extend an execution trace to not only include CPU states but also instructions: $\mathsf{tr} = (S_1, I_1, \ldots, S_T, I_T)$ where S_i is the i-th CPU state, and I_i is the i-th executed instruction. We seek an arithmetic circuit C that checks tr, in this "extended" format, for the same three properties as above: (i) *validity of instruction fetch*; (ii) *validity of instruction execution*; (iii) *validity of memory accesses*.

As in [16], checking that tr satisfies property (ii) is quite straightforward. Construct a circuit C_{exe} so that, given two CPU states S, S' and an instruction I, $C_{\mathsf{exe}}(S, S', I, g)$ is satisfied, for some guess g, if and only if S' can be reached from S, by executing I, for some state of memory. Then, C contains $T-1$ copies of C_{exe}, each wired to adjacent CPU states and an instruction, i.e., the i-th copy is $C_{\mathsf{exe}}(S_i, S_{i+1}, I_i, g_i)$.

Unlike [16], though, we verify properties (i) and (iii) jointly, via the same routing network. The auxiliary input now contains $\mathsf{tr}^* = (A_1, \ldots, A_{2T})$, purportedly equal to the memory-sorted list of *both* instructions fetches and CPU states. (Since the program $\mathbb{P}$ lies in the same read-write memory as data, an instruction fetch from $\mathbb{P}$ is merely a special type of memory load.) Thus, to check that tr satisfies properties (i) and (iii), we design C to (a) verify that $\mathsf{tr}^* = \mathsf{MemSort}(\mathsf{tr})$ via nondeterministic routing, and

(b) verify validity of all (i.e., instruction and data) memory accesses, via a new circuit C'_{mem} applied to each pair of adjacent items A_i, A_{i+1} in $\mathtt{tr}^*$. Thus, in this approach, $\mathbb{P}$ is never replicated T times; rather, the fetching of its instructions is verified together with all other memory accesses, one instruction fetch at a time.

Multiple memory-access types. Each copy of C'_{mem} inspects a pair of items in $\mathtt{tr}^*$ and (assuming $\mathtt{tr}^* = \mathsf{MemSort}(\mathtt{tr})$) must ensure consistency of "load-after-load", "load-after-store", and so on. However, unlike in [16], the byte-addressable memory of vnTinyRAM is accessed in *different-sized* blocks: instruction-size blocks for instruction fetch; word-size blocks when loading/storing words; and byte-size blocks when loading/storing bytes. The consistency checks in C'_{mem} must handle "aliasing", i.e., accesses to the same point in memory via different addresses and block sizes.

We tackle this difficulty as follows. Double-word blocks are the largest blocks in which memory is accessed (as instructions are encoded as double words; cf. Section 2.5). We thus let each item in $\mathtt{tr}^*$ always specify a double-word, even if the item's memory access was with respect to a smaller-sized block (e.g., word or byte). With this modification, we can let C'_{mem} perform consistency checks "at the double-word level", and handling word/byte accesses by mapping them to double-word accesses with suitable shifting and masking.

Booting the machine. We have so far assumed that the program $\mathbb{P}$, given as input to C, *already* appears in memory. However, the circuit C sketched so far only verifies the validity of $\mathtt{tr}$ with respect to a machine whose memory is initialized to *some* state, corresponding to the execution of *some* program. But C must verify correct execution of, specifically, $\mathbb{P}$, and so it must also verify that memory is initialized to contain $\mathbb{P}$. Since C does not explicitly maintain memory (not even the initial one) and only implicitly reasons about memory via the routing network, it is not clear how C can perform this check.

We tackle this difficulty as follows. We further modify the the execution trace $\mathtt{tr}$, by extending it with an initial *boot* section, preceding the beginning of the computation, during which the input program $\mathbb{P}$ is stored into memory, one instruction $\mathbb{P}_i$ at a time. This extends the length of both $\mathtt{tr}$ and $\mathtt{tr}^*$ from $2T$ to $\ell + 2T$, for ℓ-instruction programs, and introduces a new type of item, "boot input store", in $\mathtt{tr}^*$. Similarly, the routing network is now responsible for routing $\ell + 2T$, rather than $2T$, packets.

Further optimizations. The above construction sketch (depicted in Figure 6) is only intuitive, and does not discuss other optimizations that ultimately yield the performance that we report in Section 5.1.

For example, while [16] rely on Beneš networks, we rely on *arbitrary-size Waksman networks* [10], which only require $N(\log N - 0.91)$ switches to route N packets, instead of $2^{\lceil \log N \rceil}(\lceil \log N \rceil - 0.5)$. Besides being closer to the information-theoretic lower bound of $N(\log N - 1.443)$, such networks eliminate costly rounding effects in [16], where the size of the network is *doubled* if N is just above a power of 2.

Compiling to vnTinyRAM. To enable verification of higher-level programs, written in C, we ported the GCC compiler to the vnTinyRAM architecture, by modifying the Harvard-architecture, word-addressible TinyRAM C compiler of [16]. Given a C program, written in the same subset of C as in [16], the compiler produces the initial memory map representing a program $\mathbb{P}$. This also served to validate the vnTinyRAM architectural choices (e.g., the move to byte-addressing significantly, and added instructions, improved efficiency for many programs).

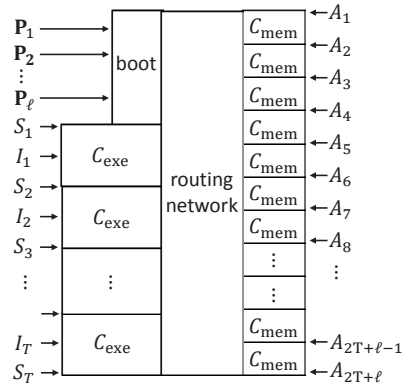

Figure 6: Outline of our universal circuit construction with the extended trace $\mathtt{tr}$ on the left and (allegedly) its memory sort $\mathtt{tr}^*$ on the right.

4 Our zk-SNARK for circuits

We discuss our second main contribution: a high-performance implementation of a zk-SNARK for arithmetic circuit satisfiability. Our approach is to *tailor* the requisite mathematical algorithms to the *specific* zk-SNARK protocol at hand. While our techniques can be instantiated in many algebraic setups and security levels, we demonstrate them in two specific settings, to facilitate comparison with prior work.

See Section 2.4 for an informal definition of a zk-SNARK for arithmetic circuit satisfiability. We improve upon and implement the zk-SNARK of Parno et al. [56]. For completeness the "PGHR protocol" is summarized in the full version of this paper, which provides pseudocode for its key generator G, prover P, and verifier V. The construction is based on QAPs, introduced in Section 2.2.

Like most other zk-SNARKs, the PGHR protocol relies on a *pairing*, which is specified by a prime $r \in \mathbb{N}$, three

cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ of order r, and a bilinear map $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. (See Section 2.3.)

A pairing is typically instantiated via a *pairing-friendly elliptic curve*. Concretely, suppose that one uses a curve E defined over $\mathbb{F}_q$, with embedding degree k with respect to r, to instantiate the pairing. Then $\mathbb{G}_T$ is set to μ_r, the subgroup of r-th roots of unity in $\mathbb{F}^*_{q^k}$. The instantiation of $\mathbb{G}_1$ and $\mathbb{G}_2$ depends on the choice of e; typically, $\mathbb{G}_1$ is instantiated as an order-r subgroup of $E(\mathbb{F}_q)$, while, for efficiency reasons [7, 8], $\mathbb{G}_2$ as an order-r subgroup of $E'(\mathbb{F}_{k/d})$ where E' is a *d-th twist* of E. Finally, the pairing e is typically a two-stage function $e(\mathscr{P}, \mathscr{Q}) := \mathsf{FE}(\mathsf{ML}(\mathscr{P}, \mathscr{Q}))$, where $\mathsf{ML}\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{F}_q^k$ is known as *Miller loop*, and $\mathsf{FE}\colon \mathbb{F}_q^k \to \mathbb{F}_q^k$ is known as *final exponentiation* and maps α to $\mathsf{FE}(\alpha) := \alpha^{(q^k-1)/r}$.

As mentioned, we instantiate our techniques based on two different curves: an Edwards curve for the 80-bit security level (as in [16]) and a Barreto–Naehrig curve for the 128-bits security level (as in [56, 27]). We selected both the Edwards curve and Barreto–Naehrig curve so that $r-1$ has high *2-adic order* (i.e., $r-1$ is divisible by a large power of 2), because this was shown to improve the efficiency of the key generator and the prover [16].

4.1 An optimized verifier

The verifier V takes as input a verification key vk, input $\vec{x} \in \mathbb{F}_r^n$, and proof π, and checks if π is a valid proof for the statement "$\vec{x} \in \mathscr{L}_C$". The computation of V consists of two parts. First, use $\mathsf{vk}_{\mathsf{IC},0}, \ldots, \mathsf{vk}_{\mathsf{IC},n} \in \mathbb{G}_1$ (part of vk) and input $\vec{x}$ to compute $\mathsf{vk}_{\vec{x}} := \mathsf{vk}_{\mathsf{IC},0} + \sum_{i=1}^{n} x_i \mathsf{vk}_{\mathsf{IC},i}$. Second, use vk, $\mathsf{vk}_{\vec{x}}$, and π, to compute 12 pairings and perform the required checks. In other words, V performs $O(n)$ scalar multiplications in $\mathbb{G}_1$, followed by $O(1)$ pairing evaluations.

With regard to V's first part, variable-base multi-scalar multiplication techniques can be used to reduce the number of $\mathbb{G}_1$ operations needed to compute $\mathsf{vk}_{\vec{x}}$ [16, 56]. With regard to V's second part, even if the pairing evaluations take constant time (independent of the input size n), these evaluations are very expensive and *dominate for small n*. Our focus here is to minimize the cost of these pairing evaluations.

When only making "black-box" use of a pairing, the verifier must evaluate 12 pairings, amounting to 12 Miller loops plus 12 final exponentiations. The straightforward approach is to compute these using a generic high-performance pairing library. We proceed differently: we obtain high-performance implementations of *subcomponents* of a pairing, and then tailor their use specifically to V's protocol.

Namely, first, we obtain state-of-the-art implementations of a Miller loop and final exponentiation. We utilize *optimal pairings* [70] to minimize the number of loop iterations in each Miller loop, and, to efficiently evaluate each Miller loop, rely on the formulas of [3] (for Edwards curves) and [20] (for BN curves). As for final exponentiation, we use multiple techniques to speed it up: [62, 44, 35, 50].

Next, building on the above foundation, we incorporate in V the following optimizations.

(1) Sharing Miller loops and final exponentiations. The verifier V computes two products of two pairings. We leverage the fact that a product of pairings can be evaluated faster than evaluating each pairing separately and then multiplying the results [60]. 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 $\mathbb{F}_{q^k}$ 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 [61]), 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.

Our implementation incorporates both of the above techniques. For example, at the 80-bit security level, separately computing 12 optimal pairings costs 13.6 ms, but the above techniques reduce the time to only 8.1 ms. We decrease this further as discussed next.

(2) 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. The other 11 pairings have one fixed input, either a generator of $\mathbb{G}_1$ or $\mathbb{G}_2$, or coming from the verification key vk.

When one input to a pairing is fixed, precomputation techniques apply [60], especially in the case when the fixed input is the *base point* in Miller's algorithm. In V, this 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. The result of the offline phase is a *processed verification key* vk^*. While vk^* is longer than vk, it allows the online phase to be faster.

E.g., at the 80-bit security level, vk^* decreases the total cost of pairing checks from 8.1 ms to 4.7 ms.

4.2 An optimized prover

The prover P takes as input a proving key pk (which includes the circuit $C\colon \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^l$), input $\vec{x} \in \mathbb{F}_r^n$, and witness $\vec{a} \in \mathbb{F}_r$. The prover P is tasked to produce a proof

π, attesting that $\vec{x} \in \mathscr{L}_C$. The computation of P consists of two main parts. First, compute the coefficients $\vec{h}$ of the polynomial $H(z) := \frac{A(z)B(z)-C(z)}{Z(z)}$, where $A,B,C \in \mathbb{F}_r[z]$ are derived from the QAP instance $(\vec{A},\vec{B},\vec{C},Z) := \mathsf{QAPinst}(C)$ and QAP witness $\vec{s} := \mathsf{QAPwit}(C,\vec{x},\vec{a})$. Second, use the coefficients $\vec{h}$, QAP witness $\vec{s}$, and public key pk to compute π.

With regard to the first part of P, the coefficients $\vec{h}$ can be efficiently computed via FFT techniques [16, 56]; our implementation follows [16], and leverages the high 2-adic order of $r-1$ for both of the elliptic curves we use. With regard to P's second part, computing π requires solving large instances of the following problem: given elements $\mathscr{Q}_1,\ldots,\mathscr{Q}_n$ all in $\mathbb{G}_1$ (or all in $\mathbb{G}_2$) and scalars $\alpha_1,\ldots,\alpha_n \in \mathbb{F}_r$, compute $\langle \vec{\alpha},\vec{\mathscr{Q}}\rangle := \alpha_1\mathscr{Q}_1 + \cdots + \alpha_n\mathscr{Q}_n$. Previous work [56, 16] has leveraged generic multi-scalar multiplication to compute π. We observe that these algorithms can be tailored to the specific scalar distributions arising in P. In P, the vector $\vec{\alpha}$ is one of two types: (i) $\vec{\alpha} \in \mathbb{F}_r^{d+1}$ and represents the coefficients of the degree-d polynomial H; or (ii) $\vec{\alpha} = (1 \circ \vec{s} \circ \delta_1 \circ \delta_2 \circ \delta_3) \in \mathbb{F}_r^{4+m}$, for random $\delta_1,\delta_2,\delta_3 \in \mathbb{F}_r$.

In case i, the entries in of $\vec{\alpha}$ are random-looking. We use the Bos–Coster algorithm [26] due to its lesser memory requirements (as compared to, e.g., [57]). We follow [19]'s suggestions and achieve an assembly-optimized heap to implement the Bos–Coster algorithm.

In case ii, 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 3). Using the above algorithm "as is" is inefficient: the algorithm works well when all the scalars have roughly the same bit complexity, but the entries in $\vec{c}$ have very different bit complexity. Indeed, $\vec{\alpha}$ equals to $\vec{s}$ augmented with a few entries; and $\vec{s}$, the QAP witness, can be thought of as the list of wire values in C when computing on $(\vec{x},\vec{a})$; the bit complexity of a wire value depends on whether it is storing a boolean value, a word value, and so on. We observe that there are only a few "types" of values, so that the entries of $\vec{\alpha}$ can be clustered into few groups of scalars with approximately the same bit complexity; we then apply the algorithm of [26] to each such group.

4.3 An optimized key generator

The key generator G takes as input a circuit $C\colon \mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^l$, and is tasked to compute a proving key pk and a verification key vk. The computation of G consists of two main parts. First, evaluate each A_i,B_i,C_i at a random element τ, where $(\vec{A},\vec{B},\vec{C},Z) := \mathsf{QAPinst}(C)$ is the QAP instance. Second, use these evaluations to compute pk and vk.

With regard to G's first part, we follow [16] and again leverage the fact that $\mathbb{F}_r$ has a primitive root of unity of large order. With regard to G's second part, it is dominated by the cost of computing pk, which requires solving large instances of the following problem: given an element $\mathscr{P}$ in $\mathbb{G}_1$ or $\mathbb{G}_2$ and scalars $\alpha_1,\ldots,\alpha_n \in \mathbb{F}_r$, compute $\alpha_1\mathscr{P},\ldots,\alpha_n\mathscr{P}$. Previous work [56, 16], used fixed-base windowing [28] to efficiently compute such fixed-base multi-scalar multiplications.

In our implementation, we achieve additional efficiency, in space rather than in time. Specifically, we leverage a structural property of QAPs derived from arithmetic circuits, in order to reduce the size of the proving key pk, as we now explain. Lemma 2.4 states that an $\mathbb{F}$-arithmetic circuit $C\colon \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$, with α wires and β gates, can be converted into a corresponding QAP of size $m = \alpha$ and degree $d \approx \beta$ over $\mathbb{F}$. Roughly, this is achieved in two steps. First, construct three matrices $\mathbf{A},\mathbf{B},\mathbf{C} \in \mathbb{F}^{(m+1)\times d}$ that encode C's topology: for each $j \in [d]$, the j-th column of $\mathbf{A},\mathbf{B}$ respectively encodes the "left" and "right" coefficients of the j-th bilinear gate in C, while the j-th column of $\mathbf{C}$ encodes the coefficients of the gate's output. Second, letting $S \subset \mathbb{F}$ be a set of size d, define $Z(z) := \prod_{\omega\in S}(z-\omega)$ and, for $i \in \{0,\ldots,m\}$, let A_i be the low-degree extension of the i-th row of $\mathbf{A}$; similarly define each B_i and C_i. All prior QAP-based zk-SNARK implementations exploit the fact that *columns* in the matrices $\mathbf{A},\mathbf{B},\mathbf{C}$ are very sparse.

In contrast, we also leverage a *different* kind of sparsity: we observe that it is common for *entire rows* of $\mathbf{A},\mathbf{B},\mathbf{C}$ to be all zeroes, causing the corresponding low-degree extensions to be zero polynomials.[3] For instance, our circuit generator typically outputs a circuit for which the percentage of non-zero polynomials in $\vec{A},\vec{B},\vec{C}$ is only about $52\%, 15\%, 71\%$ respectively. The fact that many polynomials in $\vec{A},\vec{B},\vec{C}$ evaluate to zero can be used towards reducing the size of pk, by switching from a dense representation to a sparse one.

In fact, we have *engineered* our circuit generator to reduce the number of non-zero polynomials in $\vec{B}$ as much as possible, because computations associated to evaluations of $\vec{B}$ are conducted with respect to more expensive $\mathbb{G}_2$ arithmetic, which we want to avoid as much as possible.

5 Evaluation

We evaluated our system on a desktop computer with a 3.40 GHz Intel Core i7-4770 CPU (with Turbo Boost disabled) and 32 GB of RAM. All experiments, except the largest in Figure 8 and 9, used a small fraction of the RAM. For the two largest experiments in Figure 9 we added a Crucial M4 solid state disk for swap space. (While our code supports multi-threading, our experiments are in single-thread mode, for comparison with prior work.)

[3] E.g., if the i-th wire never appears with a non-zero coefficient as the "left" input of a bilinear gate, then the i-th row of $\mathbf{A}$ is zero.

5.1 Performance of our circuit generator

In Section 3 we described our universal circuit generator; we now benchmark its performance.

Parameters. The circuit supports vnTinyRAM, which is parametrized by two quantities: the *word size* W and the *number of registers* K (see Section 2.5). We report performance for a machine with $K = 16$ registers, and two choices of word size: $W = 16$ and $W = 32$.

Methodology. Theorem 3.2 provides an asymptotic efficiency guarantee: it states that our circuit generator has efficiency $f(\ell,n,T) = O\big((\ell+n+T)\cdot\log(\ell+n+T)\big)$. To understand concrete efficiency, we "uncover" the constants hidden in the *big-oh* notation. By studying the number of gates in various subcircuits of the generated circuit $C := \mathsf{circ}(\ell,n,T)$, we computed the following (quite tight) upper bound on C's size:

$$(12+2W)\cdot\ell+(12+W)\cdot n+|C_{\mathsf{exe}}|\cdot T+(|C_{\mathsf{mem}}|+ 4\log H-1.82)\cdot H$$

where $H := (\ell+n+2T)$ is the "height" of the routing network, and

- for $(W,K) = (16,16)$: $|C_{\mathsf{exe}}| = 777$ and $|C_{\mathsf{mem}}| = 211$;
- for $(W,K) = (32,16)$: $|C_{\mathsf{exe}}| = 1114$ and $|C_{\mathsf{mem}}| = 355$.

In Figure 7, we give per-cycle gate counts (i.e., $|C|/|T|$) for various choices of (ℓ,n,T); we also give sub-counts divided among program/input boot, CPU execution, memory checking, and routing. (See the full version of this paper for an extended table with additional data.)

Discussion. We first go through the size expression, to understand it: The first two terms, $(12+2W)\cdot\ell+(12+W)\cdot n$, correspond to the pre-execution boot phase, during which an ℓ-instruction program and an n-word primary input are loaded into the machine. The term $|C_{\mathsf{exe}}|\cdot T$ corresponds to the T copies of C_{exe} used to verify each CPU transition, given the fetched instruction and two CPU states. The term $|C_{\mathsf{mem}}|\cdot H$ corresponds to the H copies of C_{mem} used to verify consistency on the memory-sorted trace. Finally, the term $(4\log H-1.82)\cdot H$ corresponds to the routing network for routing H packets (two gates for each of $(2\log H-0.91)\cdot H$ binary switches). Note that $H = (\ell+n+2T)$ because boot needs $\ell+n$ memory stores (one for each program instruction and primary input word) and execution needs $2T$ memory accesses (1 instruction fetch and 1 data store/load per execution cycle).

The gate counts in Figure 7 demonstrate the additive (instead of multiplicative) dependence on program size of our universal circuit pays off. For example, for $(W,K) = (32,16)$, a 100-fold increase in program size, from $\ell = 10^3$ to $\ell = 10^5$, barely impacts the per-cycle gate count: for $T = 2^{20}$, it increases from 1,992.5 to only 2,041.5. Indeed, the cost of program size is incurred, once and for all, during the machine boot; Figure 7 shows that the per-cycle cost of machine boot diminishes as T grows.

Second, *less than half* of C's gates are dedicated to verifying accesses to random-access memory, while the majority of gates are dedicated to verifying execution of the CPU; indeed, almost always, $|C_{\mathsf{exe}}|T > \frac{1}{2}|C|$. Put otherwise, C, which verifies an automaton with random-access memory (vnTinyRAM), has size that is less than twice that for verifying an automaton with the same CPU but no random-access memory at all. Moreover, note that the size of C_{exe} appears quite tight: for example, with $(W,K) = (32,16)$, it has size 1114, not much larger than the size of the CPU state (545 bits).

5.2 Performance of our zk-SNARK for circuit satisfiability

In Section 4 we described our zk-SNARK implementation; we now benchmark its performance.

Methodology. We provide performance characteristics for each of the zk-SNARK algorithms, G, P and V, at the 80-bit and 128-bit security levels.

(1) The key generator G takes as input an arithmetic circuit C: $\mathbb{F}_r^n \times \mathbb{F}_r^h \to \mathbb{F}_r^l$. 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). Thus, we evaluate G on a circuit with 2^i gates and 2^i wires for $i \in \{10,12,\ldots,24\}$ (and fixed $n = h = l = 100$). In Figure 8 we report the resulting running times and key sizes, as *per-gate costs*.

(2) The prover P 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 mostly depends on the number of gates and wires in C (the circuit used to generate pk); we thus evaluate P on the proving keys output by G, for the same circuits as above. In Figure 8 we report the times, as per-gate costs, and proof sizes.

(3) The verifier V takes as input a verification key vk, input $\vec{x} \in \mathbb{F}_r^n$, and proof π. Its efficiency depends only on $\vec{x}$ (since the size of $\vec{x}$ determines that of vk). Thus, we evaluate V on a random input $\vec{x} \in \mathbb{F}_r^n$ of 2^i bytes for $i \in \{2,4,\ldots,20\}$. In Figure 8 we report the resulting running times, along with corresponding key sizes.

Discussion. The data demonstrates that our zk-SNARK implementation works and scales as expected, as long as sufficient memory is available (e.g., on a desktop computer with 32GB of DRAM: up to 16 million gates). Key generation takes about 10 ms per gate of C; the size of a proving key is about 300 B per gate, and the size of a verification key is about 1 B per byte of input to C. Running the prover takes 11 ms to 14 ms per gate. For an n-byte input, proof verification time is $c_1 n + c_0$, where c_0 is a few milliseconds and c_1 is a few tenths of microseconds.

5.3 Performance of the combined system

As discussed, our circuit generator (Section 3) and zk-SNARK for circuits (Section 4) can be used in-

		Per-cycle gate count of $C := \mathsf{circ}(\ell,n,T)$ with vnTinyRAM parameters (W,K)									
		$n = 10^2, K = 16$									
		$W = 16$					$W = 32$				
		$\lvert C\rvert/T$	$\lvert C\rvert/T$ divided among				Per cycle	$\lvert C\rvert/T$ divided among			
			boot	exec.	mem.	routing		boot	exec.	mem.	routing
$\ell = 10^3$	$T = 2^{20}$	1,367.4	0.04	777.0	422.2	168.1	1,992.5	0.08	1,114.0	710.4	168.1
	$T = 2^{24}$	1,399.0	0.00	777.0	422.0	200.0	2,024.0	0.00	1,114.0	710.0	200.0
	$T = 2^{28}$	1,431.0	0.00	777.0	422.0	232.0	2,056.0	0.00	1,114.0	710.0	232.0
$\ell = 10^4$	$T = 2^{20}$	1,370.3	0.41	777.0	424.0	168.8	1,997.0	0.72	1,114.0	713.4	168.8
	$T = 2^{24}$	1,399.2	0.03	777.0	422.1	200.1	2,024.3	0.05	1,114.0	710.2	200.1
	$T = 2^{28}$	1,431.0	0.00	777.0	422.0	232.0	2,056.0	0.00	1,114.0	710.0	232.0
$\ell = 10^5$	$T = 2^{20}$	1,399.7	4.12	777.0	442.1	176.4	2,041.5	7.19	1,114.0	743.9	176.4
	$T = 2^{24}$	1,401.1	0.26	777.0	423.3	200.6	2,027.2	0.45	1,114.0	712.1	200.6
	$T = 2^{28}$	1,431.1	0.02	777.0	422.1	232.0	2,056.2	0.03	1,114.0	710.1	232.0

Figure 7: Per-cycle gate counts in $C := \mathsf{circ}(\ell,n,T)$ for different choices of (ℓ,n,T) and vnTinyRAM parameters (W,K).

		80 bits of security		128 bits of security	
key gen. G		time/$\lvert C\rvert$	$\lvert\mathsf{pk}\rvert/\lvert C\rvert$	time/$\lvert C\rvert$	$\lvert\mathsf{pk}\rvert/\lvert C\rvert$
$n = 100$	$\lvert C\rvert = 2^{10}$	0.21 ms	248.8 B	0.21 ms	304.1 B
	$\lvert C\rvert = 2^{12}$	0.16 ms	252.5 B	0.17 ms	309.1 B
	$\lvert C\rvert = 2^{14}$	0.14 ms	253.4 B	0.16 ms	310.3 B
	$\lvert C\rvert = 2^{16}$	0.12 ms	253.7 B	0.14 ms	310.6 B
	$\lvert C\rvert = 2^{18}$	0.11 ms	253.7 B	0.12 ms	310.7 B
	$\lvert C\rvert = 2^{20}$	0.10 ms	253.7 B	0.12 ms	310.7 B
	$\lvert C\rvert = 2^{22}$	0.09 ms	253.7 B	0.11 ms	310.7 B
	$\lvert C\rvert = 2^{24}$	0.08 ms	253.7 B	0.10 ms	310.7 B
	$\lvert\mathsf{vk}\rvert$	2.8 KB		3.6 KB	
prover P		time/$\lvert C\rvert$	$\lvert\pi\rvert$	time/$\lvert C\rvert$	$\lvert\pi\rvert$
$n = 100$	$\lvert C\rvert = 2^{10}$	0.18 ms	230 B	0.21 ms	288 B
	$\lvert C\rvert = 2^{12}$	0.16 ms	230 B	0.18 ms	288 B
	$\lvert C\rvert = 2^{14}$	0.14 ms	230 B	0.16 ms	288 B
	$\lvert C\rvert = 2^{16}$	0.13 ms	230 B	0.15 ms	288 B
	$\lvert C\rvert = 2^{18}$	0.12 ms	230 B	0.15 ms	288 B
	$\lvert C\rvert = 2^{20}$	0.12 ms	230 B	0.15 ms	288 B
	$\lvert C\rvert = 2^{22}$	0.11 ms	230 B	0.14 ms	288 B
	$\lvert C\rvert = 2^{24}$	0.11 ms	230 B	0.14 ms	288 B
verifier V		$\lvert\mathsf{vk}\rvert/\lvert\vec{x}\rvert$	time/$\lvert\vec{x}\rvert$	$\lvert\mathsf{vk}\rvert/\lvert\vec{x}\rvert$	time/$\lvert\vec{x}\rvert$
$\lvert\vec{x}\rvert = 4$ B		118.7 B	1.2 ms	123.4 B	1.2 ms
$\lvert\vec{x}\rvert = 16$ B		29.7 B	0.3 ms	30.8 B	0.3 ms
$\lvert\vec{x}\rvert = 64$ B		8.1 B	76.7 μs	8.7 B	81.2 μs
$\lvert\vec{x}\rvert = 256$ B		2.8 B	19.5 μs	2.9 B	20.3 μs
$\lvert\vec{x}\rvert = 1.0$ KB		1.5 B	5.4 μs	1.5 B	5.9 μs
$\lvert\vec{x}\rvert = 4.1$ KB		1.1 B	1.8 μs	1.1 B	2.1 μs
$\lvert\vec{x}\rvert = 16.4$ KB		1.1 B	0.8 μs	1.0 B	1.0 μs
$\lvert\vec{x}\rvert = 65.5$ KB		1.0 B	0.5 μs	1.0 B	0.7 μs
$\lvert\vec{x}\rvert = 262.1$ KB		1.0 B	0.4 μs	1.0 B	0.6 μs
$\lvert\vec{x}\rvert = 1.0$ MB		1.0 B	0.4 μs	1.0 B	0.5 μs

Figure 8: Per-gate costs of the key generator and prover; and per-byte costs of the verifier. ($N = 10$ and std $< 1\%$)

dependently, or combined to obtain a zk-SNARK for vnTinyRAM. For completeness, the paper's full version we spell out how these two components can be combined. Here we report measured performance of this combined system, at the 128-bit security level, and for a word size $W = 32$ and number of registers $K = 16$.

Methodology. A zk-SNARK for vnTinyRAM is a triple of algorithms $(\mathsf{KeyGen},\mathsf{Prove},\mathsf{Verify})$. Given bounds ℓ,n,T (for program size, input size, and time), the efficiency of KeyGen and Prove depends on ℓ,n,T, while that of Verify essentially depends only on ℓ,n. Thus, we benchmark the system as follows. We evaluate KeyGen and Prove for various choices of ℓ and T, while keeping $n = 100$. Instead, since the efficiency of Verify does not depend on T, we evaluate Verify, for various choices of ℓ and n, on random ℓ-instruction programs and n-word inputs. In Figure 9, we report the following measurements: KeyGen's running time, the sizes of the keys pk and vk, Prove's runtime, the (constant) proof size, and Verify's running time. For quantities growing with T, we divide by T and report the per-cycle cost.

Discussion. The measurements demonstrate that, on a desktop computer, our zk-SNARK for vnTinyRAM scales up to computations of 32,000 machine cycles, for programs with up to 10,000 instructions. Key generation takes about 200 ms per cycle; the size of a proving key is 500 KB to 650 KB per cycle, and the size of a verification key is a few kilobytes. Running the prover takes 100 ms to 200 ms per cycle. Verification times remain a few ms , even for inputs and programs of several kilobytes.

Program-specific vk**.** The time complexity of Verify is $O(\ell+n)$, so verification time grows with program size. This is inevitable, because Verify must *read* a program $\mathbb{P}$ (of at most ℓ instructions) and input $\mathbb{x}$ (of at most n words) in order to check, via the given proof π, if $(\mathbb{P},\mathbb{x}) \in \mathscr{L}_{\ell,n,T}$ (cf. Definition 2.6). However, this is inconvenient, e.g., when one has to verify many proofs relative to different inputs to the *same* program $\mathbb{P}$. In our zk-SNARK it is possible to amortize this cost as follows. Given vk and $\mathbb{P}$, one can derive, in time $O(\ell)$, a *program-specific* verification key $\mathsf{vk}_{\mathbb{P}}$, which can be used to verify proofs relative to any input to $\mathbb{P}$. Subsequently, the time complexity of Verify for any input $\mathbb{x}$ (to $\mathbb{P}$) is $O(n)$, *independent* of ℓ.

5.4 Comparison with prior work

5.4.1 Comparison with prior circuit generators

Universality is the main innovative feature of our circuit generator. *No* previous circuit generator achieves univer-

				128 bits of security				
				$W = 32, K = 16$				
				$\ell = 2$K	$\ell = 4$K	$\ell = 6$K	$\ell = 8$K	$\ell = 10$K
KeyGen	time/T	$n = 100$	$T = 4$K	209.8 ms	232.1 ms	257.5 ms	275.9 ms	306.4 ms
			$T = 8$K	190.9 ms	205.9 ms	216.1 ms	228.9 ms	238.8 ms
			$T = 16$K	195.4 ms	198.1 ms	204.2 ms	213.6 ms	218.3 ms
			$T = 32$K	206.0 ms	208.4 ms	211.2 ms	213.5 ms	223.7 ms
	|pk|/T		$T = 4$K	584.2 KB	653.6 KB	727.1 KB	784.0 KB	876.8 KB
			$T = 8$K	552.4 KB	585.2 KB	618.1 KB	655.1 KB	683.7 KB
			$T = 16$K	539.4 KB	553.9 KB	570.4 KB	586.9 KB	605.5 KB
			$T = 32$K	533.8 KB	541.1 KB	548.3 KB	555.6 KB	563.4 KB
	|vk|		$T = *$	17.0 KB	33.1 KB	49.2 KB	65.3 KB	81.5 KB
Prove	time/T	$n = 100$	$T = 4$K	75.7 ms	86.7 ms	103.4 ms	104.8 ms	133.7 ms
			$T = 8$K	69.2 ms	79.7 ms	97.0 ms	110.4 ms	113.0 ms
			$T = 16$K	89.0 ms	89.1 ms	98.4 ms	99.6 ms	103.3 ms
			$T = 32$K	98.9 ms	98.6 ms	102.3 ms	102.1 ms	114.2 ms
Verify	time	(indep. of T)	$n = 0$	19.0 ms	30.0 ms	40.6 ms	51.2 ms	61.3 ms
			$n = 10$	19.1 ms	30.2 ms	40.7 ms	51.2 ms	61.4 ms
			$n = 10^2$	19.6 ms	30.7 ms	41.3 ms	51.8 ms	61.9 ms
			$n = 10^3$	23.0 ms	34.1 ms	44.7 ms	55.2 ms	65.4 ms
			$n = 10^4$	48.9 ms	60.0 ms	70.6 ms	81.1 ms	91.3 ms

Figure 9: Per-cycle costs of KeyGen and Prove for various program sizes ℓ, and total running time of Verify for various ℓ and n.

sality. (See Figure 1 and Section 3.)

Putting universality aside and focusing on efficiency instead, a comparison with previous circuit generators is a multi-faceted problem. On one hand, due to a shared core of techniques, a comparison with [16]'s circuit generator is straightforward, and shows significant improvements in circuit size, especially as program size grows. See Section 1.4.1 and Figure 2 (the figure is for $W, K = 16$).

Instead, a comparison with other circuit generators [66, 64, 56, 27] is complex. First, they support a smaller class of programs (see Figure 1), so a programmer must "write around" the limited functionality, somehow. And second, their efficiency is not easily specified: due to the program-analysis techniques (see Section 3.1) the output circuit is ad hoc for the given program, and the only way to know its size is to actually run the circuit generator.

Compared to [66, 64, 56, 27], our circuit generator performs better for programs that are rich in memory accesses and control flow, and worse for programs that are more "circuit like".

Comparison with [66, 64, 56]. The circuit generators in [66, 64, 56] restrict loop iteration bounds and memory accesses to be known at compile time; if a program does not respect these restrictions, it must be first somehow mapped to another one that does. For simplicity, we take [56]'s circuit generator (the latest one) as representative and, to illustrate the differences between [56]'s and our circuit generator, we consider two "extremes".

On one extreme, we wrote a simple C program multiplying two 10×10 matrices of 16-bit integers. The circuit generator in [56] produces a circuit with 1100 gates; instead, our circuit generator (when given the corresponding vnTinyRAM assembly) produces a much larger circuit: one with $\approx 10^7$ gates.

On the other extreme, we consider a program making many random accesses to memory: pointer-chasing. Given a permutation π of $[N]$, start position $i \in [N]$, and an integer k, the program outputs $\pi^k(i)$, the element obtained by starting from i and following "pointers" for k times. Since no information about π is known at compile time, the only way of obtaining $\pi(j)$, the pointer to follow, in [56] is via a *linear scan*. On a simple C program that does one linear scan of π to obtain each new pointer, [56]'s generator outputs a circuit with $2Nk+1$ gates (each of the k array accesses costs $2N$ gates).

In vnTinyRAM, the corresponding program $\mathbb{P}$ consists of 9 instructions, and the input $\mathtt{x}$ to it is $N+3$ words. Booting vnTinyRAM with $\mathbb{P}$ and $\mathtt{x}$ requires $9+N+3$ "boot stores" (see Section 3.2), and takes $5+4k$ cycles to execute (independent of N). Say that we fix $k = 10$; then, in our circuit generator (with $W = 32$ and $K = 16$), each cycle costs about 2000 gates, and can perform a random access to memory. Thus, pointer chasing in our case is cheaper than in [56] already for $N > 5000$, and the multiplicative saving, which is about $\frac{20N}{2000 \cdot (5+40)} = \frac{N}{4500}$, grows unbounded as N increases.

Comparison with [27]. The circuit generator of [27] is also based on program analysis, but provides an additional feature that allows data-dependent memory accesses: a program may access memory by guessing the value and verifying its validity via a subcircuit that checks Merkle-tree authentication paths. In [27], memory consists of 2^{30} cells, and each access costs many gates: 140K for a load, and 280K for a store. In comparison, in our circuit generator for vnTinyRAM (with word size $W = 32$ so that memory has 2^{32} cells), each memory store/load costs less than 1000 gates out of about 2000 per cycle (see Section 5.1). Besides the aforementioned feature, [27] rely on program analysis, and (as in [66, 64, 56]) only support bounded control flow. Thus, [27] performs better than our circuit generator for programs with bounded control flow and few data-dependent accesses to memory.

5.4.2 Comparison with prior zk-SNARKs

Addressing the other component of our system, the zk-SNARK for circuits: Figure 3 compares our implementation with prior ones, on a 1-million-gate circuit with a 1000-byte input. As shown, we mildly improve the key generation time and, more importantly, significantly improve the "online" costs of proving and verification.

6 Conclusion

We have presented two main contributions: (i) a circuit generator for a von Neumann RISC architecture that is *universal* and scales *additively* with program size; and (ii) a high-performance zk-SNARK for arithmetic circuit satisfiability. These two components can be used independently to the benefit of other systems, or combined into a zk-SNARK that can prove/verify correctness of computations on this architecture.

The benefits of universality. Universality attains the conceptual advance of *once-and-for-all key generation*, allowing verifying all programs up to a given size. This removes major issues in prior systems: expensive per-program key generation and the thorny issue of conducting it anew in a trusted way for every program.

The price of universality. The price of universality is still very high. Going forward, and aiming for widespread use in security applications, more work is required to slash costs of key generation and proving so to scale up to larger computations: e.g., billion-gate circuits, or millions of vnTinyRAM cycles, and beyond. An interesting open problem is whether the "program analysis" techniques underlying most prior circuit generators [66, 64, 56, 27], typically more efficient for restricted classes of programs, can be used to construct universal circuits.

Beyond vnTinyRAM. Finally, going beyond the foundation of a von Neumann RISC architecture, more work lies ahead towards a richer architecture (e.g., efficient support for floating-point arithmetic and cryptographic acceleration), code libraries, and tighter compilers.

A Other prior work

Prior work most relevant to us is about zk-SNARKs, and 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. [42] obtained an interactive proof for outsourcing computations of *low-depth circuits*. A set of works [32, 68, 67] has optimized and implemented the protocol of [42]. The protocol of [42] can also be reduced to a two-message argument system [48, 47]. Canetti et al. [30] showed how to extend the techniques in [42] to also handle non-uniform circuits.

Batching arguments. Ishai et al. [46] constructed a *batching argument* for NP, where, to simultaneously verify that N circuits of size S are satisfiable, the verifier runs in time $\max\{S^2, N\}$.

A set of works [63, 65, 66, 64] has improved, optimized, and implemented the batching argument of Ishai et al. [46] for the purpose of outsourcing computation. In particular, by relying on quadratic arithmetic programs of [38], Setty et al. [64] have improved the running time of the verifier and prover to $\max\{S,N\} \cdot \text{poly}(\lambda)$ and $\tilde{O}(S) \cdot \text{poly}(\lambda)$ respectively. Vu et al. [71] provide a system that incorporates both the batching arguments of [63, 65, 66, 64] as well as the interactive proofs of [32, 68, 67]. The system decides which of the two approaches is more efficient to use for outsourcing a given computation.

Braun et al. [27] 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. [29] 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 [34]. The protocol in [29] 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 addresses the problem of translating high-level languages into low-level languages such as circuits. Most prior work only supports restricted classes of programs: [66, 64] present a circuit generator based on Fairplay [53, 12], whose SFDL language does not support important primitives and has inefficient support for others; [56] present a circuit generator for programs without data dependencies (pointers and array indices must be known at compile time, and so do loop iteration bounds).

Other works support more general functionality: [16] rely on nondeterministic routing to support random-access machine computations [14]; [27] rely on online memory checking [24, 14] to support accessing untrusted storage from a circuit. See [27, Section 2] for a more detailed overview of some of the above techniques.

Other cryptographic tools. *Fully-homomorphic encryption* (FHE) [39] and *probabilistically-checkable proofs* [5, 4] are powerful tools that are often used in protocols for outsourcing computations (with integrity or confidentiality guarantees, or both) [49, 54, 2, 37, 31, 47, 41]. However, such constructions have so far not been explored in practice. Another powerful tool is *secure multi-party computation* [40, 13], but most work in this area does not consider the goal of succinctness.

B Case study: `memcpy`

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 unknown, and `memcpy` is invoked as a subroutine in a larger program, the trivial solution *does not work*, and an efficient implementation is needed.

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). But, in vnTinyRAM, one can do better: loop unrolling can be used to avoid paying for the 2 "control" instructions. Asymptotically, the optimal number of unrollings *depends* on the array length: it is $\Theta(\sqrt{m})$. Thus, optimal unrolling requires dynamic code generation on 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 $m \geq 3000$.

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.

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, Technology and Space; the Simons Foundation, with a Simons Award for Graduate Students in Theoretical Computer Science; and the Skolkovo Foundation.

References

[1] AJTAI, M., KOMLÓS, J., AND SZEMERÉDI, E. An $o(n \log n)$ sorting network. In *STOC '83* (1983).

[2] APPLEBAUM, B., ISHAI, Y., AND KUSHILEVITZ, E. From secrecy to soundness: Efficient verification via secure computation. In *ICALP '10* (2010).

[3] ARÈNE, C., LANGE, T., NAEHRIG, M., AND RITZENTHALER, C. Faster computation of the Tate pairing. *Journal of Number Theory* (2011).

[4] ARORA, S., LUND, C., MOTWANI, R., SUDAN, M., AND SZEGEDY, M. Proof verification and the hardness of approximation problems. *JACM* (1998).

[5] ARORA, S., AND SAFRA, S. Probabilistic checking of proofs: a new characterization of NP. *JACM* (1998).

[6] BABAI, L., FORTNOW, L., LEVIN, L. A., AND SZEGEDY, M. Checking computations in polylogarithmic time. In *STOC '91* (1991).

[7] BARRETO, P. S. L. M., KIM, H. Y., LYNN, B., AND SCOTT, M. Efficient algorithms for pairing-based cryptosystems. In *CRYPTO '02* (2002).

[8] BARRETO, P. S. L. M., LYNN, B., AND SCOTT, M. Efficient implementation of pairing-based cryptosystems. *Journal of Cryptology* (2004).

[9] BARRETO, P. S. L. M., AND NAEHRIG, M. Pairing-friendly elliptic curves of prime order. In *SAC'05* (2006).

[10] BEAUQUIER, B., AND ÉRIC, D. On arbitrary size Waksman networks and their vulnerability. *Parallel Processing Letters* (2002).

[11] BELLARE, M., AND GOLDREICH, O. On defining proofs of knowledge. In *CRYPTO '92* (1993).

[12] BEN-DAVID, A., NISAN, N., AND PINKAS, B. FairplayMP: a system for secure multi-party computation. In *CCS '08* (2008).

[13] BEN-OR, M., GOLDWASSER, S., AND WIGDERSON, A. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC '88* (1988).

[14] BEN-SASSON, E., CHIESA, A., GENKIN, D., AND TROMER, E. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *ITCS '13* (2013).

[15] BEN-SASSON, E., CHIESA, A., GENKIN, D., AND TROMER, E. On the concrete efficiency of probabilistically-checkable proofs. In *STOC '13* (2013).

[16] BEN-SASSON, E., CHIESA, A., GENKIN, D., TROMER, E., AND VIRZA, M. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO '13* (2013).

[17] BEN-SASSON, E., CHIESA, A., GENKIN, D., TROMER, E., AND VIRZA, M. TinyRAM architecture specification v2.00, 2013.

[18] BEN-SASSON, E., GOLDREICH, O., HARSHA, P., SUDAN, M., AND VADHAN, S. Short PCPs verifiable in polylogarithmic time. In *CCC '05* (2005).

[19] BERNSTEIN, D. J., DUIF, N., LANGE, T., SCHWABE, P., AND YANG, B.-Y. High-speed high-security signatures. In *CHES '11* (2011).

[20] BEUCHAT, J.-L., GONZÁLEZ-DÍAZ, J. E., MITSUNARI, S., OKAMOTO, E., RODRÍGUEZ-HENRÍQUEZ, F., AND TERUYA, T. High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves. In *Pairing '10* (2010).

[21] BITANSKY, N., CANETTI, R., CHIESA, A., AND TROMER, E. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *STOC '13* (2013).

[22] BITANSKY, N., CHIESA, A., ISHAI, Y., OSTROVSKY, R., AND PANETH, O. Succinct non-interactive arguments via linear interactive proofs. In *TCC '13* (2013).

[23] BLUM, M., DE SANTIS, A., MICALI, S., AND PERSIANO, G. Non-interactive zero-knowledge. *SIAM J. Comp.* (1991).

[24] BLUM, M., EVANS, W., GEMMELL, P., KANNAN, S., AND NAOR, M. Checking the correctness of memories. In *FOCS '91* (1991).

[25] BLUM, M., FELDMAN, P., AND MICALI, S. Non-interactive zero-knowledge and its applications. In *STOC '88* (1988).

[26] BOS, J., AND COSTER, M. Addition chain heuristics. In *CRYPTO '89* (1989).

[27] BRAUN, B., FELDMAN, A. J., REN, Z., SETTY, S., BLUMBERG, A. J., AND WALFISH, M. Verifying computations with state. In *SOSP '13* (2013).

[28] BRICKELL, E. F., GORDON, D. M., MCCURLEY, K. S., AND WILSON, D. B. Fast exponentiation with precomputation. In *EUROCRYPT '92* (1993).

[29] CANETTI, R., RIVA, B., AND ROTHBLUM, G. N. Practical delegation of computation using multiple servers. In *CCS '11* (2011).

[30] CANETTI, R., RIVA, B., AND ROTHBLUM, G. N. Two protocols for delegation of computation. In *ICITS 12* (2012).

[31] CHUNG, K.-M., KALAI, Y., AND VADHAN, S. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO '10* (2010).

[32] CORMODE, G., MITZENMACHER, M., AND THALER, J. Practical verified computation with streaming interactive proofs. In *ITCS '12* (2012).

[33] EDWARDS, H. M. A normal form for elliptic curves. *Bulletin of the American Mathematical Society* (2007).

[34] FEIGE, U., AND KILIAN, J. Making games short. In *STOC '97* (1997).

[35] FUENTES-CASTAÑEDA, L., KNAPP, E., AND RODRÍGUEZ-HENRÍQUEZ, F. Faster hashing to $\mathbb{G}_2$. In *SAC '11* (2012).

[36] GAL, A., EICH, B., SHAVER, M., ANDERSON, D., MANDELIN, D., HAGHIGHAT, M. R., KAPLAN, B., HOARE, G., ZBARSKY, B., ORENDORFF, J., RUDERMAN, J., SMITH, E. W., REITMAIER, R., BEBENITA, M., CHANG, M., AND FRANZ, M. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09* (2009).

[37] GENNARO, R., GENTRY, C., AND PARNO, B. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *CRYPTO '10* (2010).

[38] GENNARO, R., GENTRY, C., PARNO, B., AND RAYKOVA, M. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT '13* (2013).

[39] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *STOC '09* (2009).

[40] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC '87* (1987).

[41] GOLDWASSER, S., KALAI, Y., POPA, R. A., VAIKUNTANATHAN, V., AND ZELDOVICH, N. Reusable garbled circuits and succinct functional encryption. In *STOC '13* (2013).

[42] GOLDWASSER, S., KALAI, Y. T., AND ROTHBLUM, G. N. Delegating computation: Interactive proofs for Muggles. In *STOC '08* (2008).

[43] GOLDWASSER, S., MICALI, S., AND RACKOFF, C. The knowledge complexity of interactive proof systems. *SIAM J. Comp.* (1989).

[44] GRANGER, R., AND SCOTT, M. Faster squaring in the cyclotomic subgroup of sixth degree extensions. In *PKC'10* (2010).

[45] GROTH, J. Short non-interactive zero-knowledge proofs. In *ASIACRYPT '10* (2010).

[46] ISHAI, Y., KUSHILEVITZ, E., AND OSTROVSKY, R. Efficient arguments without short PCPs. In *CCC '07* (2007).

[47] KALAI, Y., RAZ, R., AND ROTHBLUM, R. Delegation for bounded space. In *STOC '13* (2013).

[48] KALAI, Y. T., AND RAZ, R. Probabilistically checkable arguments. In *CRYPTO '09* (2009).

[49] KILIAN, J. A note on efficient zero-knowledge proofs and arguments. In *STOC '92* (1992).

[50] KIM, T., KIM, S., AND CHEON, J. H. On the final exponentiation in Tate pairing computations. *IEEE Trans. on Inf. Theory* (2013).

[51] LIPMAA, H. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *TCC '12* (2012).

[52] LIPMAA, H. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In *ASIACRYPT '13* (2013).

[53] MALKHI, D., NISAN, N., PINKAS, B., AND SELLA, Y. Fairplay — a secure two-party computation system. In *SSYM '04* (2004).

[54] MICALI, S. Computationally sound proofs. *SIAM J. Comp.* (2000).

[55] NAOR, M., AND YUNG, M. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *STOC '90* (1990).

[56] PARNO, B., GENTRY, C., HOWELL, J., AND RAYKOVA, M. Pinocchio: Nearly practical verifiable computation. In *Oakland '13* (2013).

[57] PIPPENGER, N. On the evaluation of powers and monomials. *SIAM J. Comp.* (1980).

[58] RIGO, A., AND PEDRONI, S. PyPy's approach to virtual machine construction. In *OOPSLA '06* (2006).

[59] SCOTT, M. Computing the Tate pairing. In *CT-RSA '05* (2005).

[60] SCOTT, M. Implementing cryptographic pairings. In *Pairing '07* (2007).

[61] SCOTT, M., AND BARRETO, P. S. L. M. Compressed pairings. In *CRYPTO '04* (2004).

[62] SCOTT, M., BENGER, N., CHARLEMAGNE, M., DOMINGUEZ PEREZ, L. J., AND KACHISA, E. J. On the final exponentiation for calculating pairings on ordinary elliptic curves. In *Pairing '09* (2009).

[63] SETTY, S., BLUMBERG, A. J., AND WALFISH, M. Toward practical and unconditional verification of remote computations. In *HotOS '11* (2011).

[64] SETTY, S., BRAUN, B., VU, V., BLUMBERG, A. J., PARNO, B., AND WALFISH, M. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys '13* (2013).

[65] SETTY, S., MCPHERSON, M., BLUMBERG, A. J., AND WALFISH, M. Making argument systems for outsourced computation practical (sometimes). In *NDSS '12* (2012).

[66] SETTY, S., VU, V., PANPALIA, N., BRAUN, B., BLUMBERG, A. J., AND WALFISH, M. Taking proof-based verified computation a few steps closer to practicality. In *Security '12* (2012).

[67] THALER, J. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO '13* (2013).

[68] THALER, J., ROBERTS, M., MITZENMACHER, M., AND PFISTER, H. Verifiable computation with massively parallel interactive proofs. *CoRR* (2012).

[69] VALIANT, P. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In *TCC '08* (2008).

[70] VERCAUTEREN, F. Optimal pairings. *IEEE Trans. on Inf. Theory* (2010).

[71] VU, V., SETTY, S., BLUMBERG, A. J., AND WALFISH, M. A hybrid architecture for interactive verifiable computation. In *Oakland '13* (2013).

[72] ZAHUR, S., AND EVANS, D. Circuit structures for improving efficiency of security and privacy tools. In *SP '13* (2013).