SNARKs for C : Verifying Program Executions Succinctly and in Zero Knowledge

Eli Ben-Sasson[1], Alessandro Chiesa[2], Daniel Genkin[2],
Eran Tromer[3], and Madars Virza[2]

[1] Technion
{eli,danielg3}@cs.technion.ac.il
[2] MIT
{alexch,madars}@csail.mit.edu
[3] Tel Aviv University
tromer@cs.tau.ac.il

Abstract. An argument system for NP is a proof system that allows efficient verification of NP statements, given proofs produced by an untrusted yet computationally-bounded prover. Such a system is non-interactive and publicly-verifiable if, after a trusted party publishes a proving key and a verification key, anyone can use the proving key to generate non-interactive proofs for adaptively-chosen NP statements, and proofs can be verified by anyone by using the verification key.

We present an implementation of a publicly-verifiable non-interactive argument system for NP. The system, moreover, is a zero-knowledge proof-of-knowledge. It directly proves correct executions of programs on TinyRAM, a nondeterministic random-access machine tailored for efficient verification. Given a program P and time bound T, the system allows for proving correct execution of P, on any input x, for up to T steps, after a one-time setup requiring $\tilde{O}(|P| \cdot T)$ cryptographic operations. An honest prover requires $\tilde{O}(|P| \cdot T)$ cryptographic operations to generate such a proof, while proof verification can be performed with only $O(|x|)$ cryptographic operations. This system can be used to prove the correct execution of C programs, using our TinyRAM port of the GCC compiler.

This yields a zero-knowledge Succinct Non-interactive ARgument of Knowledge (zk-SNARK) for program executions, in the preprocessing model — a powerful solution for delegating NP computations, with several features not achieved by previously-implemented primitives.

Our approach builds on recent theoretical progress in the area. We present efficiency improvements and implementations of two main ingredients:

1. Given a C program, we produce a circuit whose satisfiability encodes the correctness of execution of the program. Leveraging nondeterminism, the generated circuit's size is merely quasilinear in the size of the computation. In particular, we efficiently handle arbitrary and data-dependent loops, control flow, and memory accesses. This is in contrast with existing "circuit generators", which in the general case produce circuits of quadratic size.
2. Given a linear PCP for verifying satisfiability of circuits, we produce a corresponding SNARK. We construct such a linear PCP (which, moreover, is zero-knowledge and very efficient) by building and improving on recent work on quadratic arithmetic programs.

R. Canetti and J.A. Garay (Eds.): CRYPTO 2013, Part II, LNCS 8043, pp. 90–108, 2013.

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

1 Introduction

Proof systems for NP let an untrusted prover convince a verifier that "$x \in L$" where L is some fixed NP-complete language. Proof systems for NP that satisfy the *zero knowledge* and *proof of knowledge* properties are a powerful tool that enables a party to prove that he or she "knows" a secret satisfying certain properties, without revealing anything about the secret itself. Such proofs are important building blocks of many cryptographic tools, including secure computation [GMW87, BGW88], group signatures [BW06, Gro06], malleable proof systems [CKLM12], anonymous credentials [BCKL08], delegatable credentials [BCC$^+$09], electronic voting [KMO01, Gro05, Lip11], and many others. Known constructions of zero-knowledge proofs of knowledge are practical only when proving statements of special form that avoid generic NP reductions (e.g., proving pairing-product equations [Gro06]). Obtaining implementations that are both generic and efficient in practice is a long-standing goal in cryptography [BBK$^+$09, ABB$^+$12].

Due to differences in computational power among parties, many applications also require *succinct verification*: the verifier is able to check a nondeterministic polynomial-time computation in time that is much shorter than the time required to run the computation when given a valid NP witness. For instance, this is the case when a weak client wishes to *outsource* (or *delegate*) a computation to an untrusted worker. The additional requirement of succinct verification has still not been achieved in practice in its full generality, despite recent theoretical and practical progress.

Furthermore, a difficulty that arises when studying the efficiency of proofs for *arbitrary* NP statements is the problem of *representation*. Proof systems are typically designed for inconvenient NP-complete languages such as circuit satisfiability or algebraic constraint satisfaction problems, while in practice, many of the problem statements we are interested in proving are easiest to express via algorithms written in some high-level programming language. Modern compilers can efficiently transform these algorithms into a program to be executed on a random-access machine (RAM). Therefore, we seek proof systems that efficiently support NP statements expressed as the correct execution of a RAM program.

1.1 Succinct Verification in the Preprocessing Model

There has been a lot of work on the problem of how to enable a verifier to succinctly verify long computations. Depending on the model, the functionality, and the security notion, different constructions are known. See the extended version of this paper for a brief summary of prior theoretical work in this area.

Many constructions achieving some form of succinct verification are only computationally sound: their security is based on cryptographic assumptions, and therefore are secure only against bounded-size provers. Yet, computational soundness seems inherent in many of these cases [BHZ87, GH98, GVW02, Wee05]. Proofs (interactive or non) that are only computationally sound are also known as *arguments* [BCC88].

In this work we are interested in non-interactive succinct verification in the preprocessing model: we investigate efficient implementations of *succinct non-interactive arguments (SNARGs) in the preprocessing model*. Also, we focus on the *publicly-verifiable* case, where a non-interactive proof can be (succinctly) verified by anyone. For simplicity, we start by introducing this cryptographic primitive for circuit satisfiability: the *circuit satisfaction problem* of a circuit $C\colon \{0,1\}^n \times \{0,1\}^h \to \{0,1\}$ is the relation $\mathcal{R}_C = \{(x,a) \in \{0,1\}^n \times \{0,1\}^h : C(x,a) = 1\}$; its language is $\mathcal{L}_C = \{x \in \{0,1\}^n : \exists\, a \in \{0,1\}^h,\ C(x,a) = 1\}$.

A ***publicly-verifiable preprocessing SNARG*** (or, for brevity in this paper, simply ***SNARG***) is a triple of algorithms (G, P, V), respectively called *key generator*, *prover*, and *verifier*, working as follows. The (probabilistic) key generator G, on input a security parameter λ and circuit $C\colon \{0,1\}^n \times \{0,1\}^h \to \{0,1\}$, outputs a *proving key* σ and a *verification key* τ; these are the system's public parameters, which need to be generated only once per circuit. After that, anyone can use the proving key σ to generate non-interactive proofs for the language $\mathcal{L}_C$, and anyone can use the verification key τ to check these proofs. Namely, given σ and any $(x,a) \in \mathcal{R}_C$, the honest prover $P(\sigma, x, a)$ produces a proof π attesting that $x \in \mathcal{L}_C$; the verifier $V(\tau, x, \pi)$ checks that π is a valid proof for $x \in \mathcal{L}_C$.

The efficiency requirements are as follows:

- running the generator G on input $(1^\lambda, C)$ requires $\mathrm{poly}(|C|)$ cryptographic operations;
- running the prover P on input (σ, x, a) also requires $\mathrm{poly}(|C|)$ cryptographic operations; but
- running the verifier V on input (τ, x, π) only requires $\mathrm{poly}(|x|)$ cryptographic operations; and
- an honestly-generated (publicly-verifiable non-interactive) proof has size $\mathrm{poly}(\lambda)$.

We require (adaptive) computational soundness: for every polynomial-size prover P^*, constant $c > 0$, large enough security parameter $\lambda \in \mathbb{N}$, and circuit $C\colon \{0,1\}^n \times \{0,1\}^h \to \{0,1\}$ of size λ^c, letting $(\sigma, \tau) \leftarrow G(1^\lambda, C)$, if $P^*(\sigma, \tau)$ outputs an adaptively-chosen (x, π) such that there is no a for which $(x,a) \in \mathcal{R}_C$ then $V(\tau, x, \pi)$ rejects (except with negligible probability over G's randomness).

Furthermore, if a SNARG satisfies a certain natural proof-of-knowledge property, we call it a *SNARK*. If it additionally satisfies a certain natural zero-knowledge property, we call it a *zero-knowledge SNARK* (zk-SNARK). See the extended version of this paper for definitions.

1.2 Motivation

It would be wonderful to have efficient and generic implementations of SNARGs *without* any expensive preprocessing. (That is, running the generator G only requires $\mathrm{poly}(\lambda)$ time instead of $\mathrm{poly}(|C|)$ cryptographic operations.) The two known approaches to constructing such SNARGs are Micali's "computationally-sound proofs" [Mic00], and the bootstrapping techniques of Bitansky et al. [BCCT13]. Algorithmically, both are complex (and, thus far, expensive) constructions: the former requires probabilistically-checkable proofs (PCPs) [BFLS91] and the latter the use of recursive proof-composition.

Thus, even in light of recent advances in the computational efficiency of PCPs [BS08, Din07, MR08, BCGT13b], it seems wise to first investigate efficient implementations of SNARGs in the preprocessing model, which is a less demanding model because it allows G to conduct a one-time expensive computation "as a setup phase". Despite the expensive preprocessing, this model is potentially useful for many applications: while the generator G does require a lot of work to set up the system's public parameters (which only depend on the given circuit C but not the input to C), this work can be subsequently amortized over many succinct proof verifications (where each proof is with respect to a new, adaptively-chosen, input to C).

In this work we focus on the preprocessing model, due to the simpler and tighter constructions known in it. Recent works [Gro10, Lip12, GGPR13, BCI$^+$13] constructed zk-SNARKs based on knowledge-of-exponent assumptions [Dam92, HT98, BP04] in bilinear groups, and all of these constructions achieved the attractive feature of having proofs consisting of only $O(1)$ group elements and of having verification via simple arithmetic circuits that are linear in the size of the input for the circuit.

In this vein, Bitansky et al. [BCI$^+$13] gave a general technique for constructing zk-SNARKs. First, they define a *linear PCP* to be one where the honest proof oracle is a linear function (over an underlying field), and soundness is required to hold *only* for linear proof oracles. Then, they show a transformation (also based on knowledge-of-exponent assumptions) from any linear PCP with a *low-degree verifier* to a SNARK; also, if the linear PCP is honest-verifier zero-knowledge (HVZK), then the resulting SNARK is zero knowledge.

When combining with other works the transformation of Bitansky et al. from linear PCPs, one obtains a theoretically simple and attractive route for the construction of zk-SNARKs. Specifically, the work on *quadratic-span programs* (QSPs) and *quadratic arithmetic programs* (QAPs) of Gennaro et al. [GGPR13] implies efficient constructions of (HVZK) linear PCPs with low-degree verifiers for circuit satisfiability, and the work on fast reductions of Ben-Sasson et al. [BCGT13a] implies that random-access machine computations can be efficiently reduced to circuit satisfiability.

In this paper, we study the tantalizing question of whether the aforementioned theoretical progress can be translated into efficient implementations of zk-SNARKs. As always, bringing theory to practice requires significant additional insights and improvements, and tackling these is the goal of our work.

1.3 Contributions

In this work we present an implementation of a zk-SNARK (i.e., a non-interactive argument system for NP with the properties of zero knowledge, proof of knowledge, and succinct verification in the preprocessing model). Moreover, our implementation efficiently supports NP statements expressed as the correct execution of a program on a nondeterministic random-access machine or (via a compiler we wrote) expressed as the correct execution of a C program. Our contributions can be summarized as follows:

1) Verifying circuit satisfiability via linear PCPs. We obtain an implementation of zk-SNARKs for (arithmetic) circuit satisfiability with excellent asymptotic efficiency: linear-time generator, quasilinear-time prover, and linear-time verifier. Moreover, proofs consist of only 12 group elements (a total of 780 bytes), independently of the circuit C or the input x to C. A proof provides 128 bits of security.

Our approach consists of two steps. First, we significantly optimized and implemented the transformation of Bitansky et al. [BCI$^+$13]; our optimizations rely on multi-exponentiation algorithms (see [Ber02] and references therein) and parallelism. Second, by building on the work on quadratic arithmetic programs (QAPs) of Gennaro et al. [GGPR13] and by leveraging algebraic structure of a carefully-chosen field, we give an efficient implementation of a linear PCP with a low-degree verifier. When verifying that $x \in \mathcal{L}_C$, our linear PCP has 5 queries of $2|C|$ field elements each; each query can be generated in linear time; the prover can compute the linear proof oracle via an arithmetic circuit of size $O(|C| \log |C|)$ and depth $O(\log |C|)$; the answers to the 5 queries can be verified with $O(|x|)$ field operations.

2) From correctness of program execution to circuit satisfiability. The SNARKs generated by the previous transformation are for proving the satisfiability of a given (arithmetic) circuit. However, programs are easier to write using high-level programming languages, like C, and it is often not realistic to require an arbitrary application to already provide a circuit encoding the NP statement of interest. We address this problem by providing a "circuit generator" that differs significantly and qualitatively from *all* previous implementations of circuit generators (e.g., Fairplay [MNPS04, BDNP08]): it leverages nondeterminism to reduce the size of the output circuit. Specifically, previous circuit generators produce circuits of $O(T^2)$ size for T-step computations in the worst case, whereas our generator produces circuits of only $O(T \log T)$ size. In more detail, our solution to the circuit generation problem is as follows:

(i) We design a minimalistic nondeterministic random-access machine, called TinyRAM.

(ii) We obtain a transformation, significantly more efficient than the one in [BCGT13a], that takes as input a TinyRAM program $\mathbf{P}$ and a time bound T and outputs a circuit whose satisfiability encodes the correct execution of $\mathbf{P}$ for up to T steps. Our efficiency improvements are achieved by leveraging field operations and nondeterminism in order to verify several types of crucial (boolean) computations via smaller arithmetic circuits. We implemented our transformation.

(iii) We complement the above transformation with a GCC backend, for compiling programs written in a subset of C into TinyRAM assembly. This compiler provides a convenient way to obtain TinyRAM programs for problems of interest. Crucially, we can efficiently support arbitrary and data-dependent loops, control flow, and memory accesses.

Our choice of architecture for TinyRAM strikes a balance between the ability to efficiently compile programs into TinyRAM assembly code, and the need to design small circuits for the transition function of TinyRAM.

Delegation for NP Programs. Together, our contributions yield a system to verify program executions succinctly and in zero knowledge.

In particular, our contributions provide a solution for non-interactively delegating arbitrary NP computations, also in a way that does not compromise the privacy of any input that the untrusted worker contributes to the computation. Previous implementation work did not achieve many of the features enjoyed by our implementation. (See the extended version of this paper for a comparison with prior implementation work.)

2 From Correctness of Program Execution to Circuit Satisfiability

As summarized in Section 1.3, we implemented an *efficient* transformation that reduces correctness of program execution to circuit satisfiabiliy. The following gives further design and performance details about this transformation. Concretely, in Section 2.1 we motivate and discuss our choice of architecture, TinyRAM. Then, in Section 2.2, we discuss implementation and performance of our compiler from C to TinyRAM assembly. Finally, in Section 2.3, we discuss implementation and performance of our reduction from the correctness of TinyRAM assembly to circuit satisfiability.

2.1 The TinyRAM Architecture

To reason about correctness of program executions, we first need to fix a specific random-access machine. An attractive choice is to pick the instruction set architecture (ISA) of some existing, well-supported family of CPUs (e.g., x86 or ARM). We could then reuse existing tools and software written for those CPUs. This is possible in principle.

However, the design of CPUs typically focuses on efficient ways of getting data and code, at the right time, to the different executions units of the CPU, with the goal of maximizing utilization of these units. This is achieved by complex mechanisms whose size can dwarf the *functional core* circuitry (execution units, register file, instruction decoding, and so on). Thus, modern CPUs afford, and employ, large and rich instruction sets. As explained next, the efficiency considerations are very different in our context.

Executing vs. Verifying. CPUs and their ISAs are optimized for fast *execution* of programs. However, we are interested in fast *verification* of (alleged) past executions. In our setting, the computation *has already been executed* and we possess a trace of this execution, giving the state of the processor (registers and flags) at every time step. Our goal is to efficiently verify the correctness of the trace: that every state in the trace follows from the preceding one.

This means that values that are expensive to produce during the execution become readily available for verification in the trace. For example, in real CPUs, reading from external memory is relatively slow and a large fraction of the circuitry is dedicated to caching data. However, in the trace, the result of a load from memory is readily seen in the processor state at the end of the computation step; thus the need for caches is moot. Similarly, modern CPUs use complicated speculative-execution and branch-prediction mechanisms to keep their execution pipelines full; but a trace verifier going down the trace can "peek into the future" and readily observe control flow.

The elimination of the above mechanisms, and many others, affects the ISA. In particular, it means that the aforementioned functional core circuitry dominates cost. This leads to the next consideration.

Transition Function Complexity. We are ultimately interested in carrying out the verification of a trace via a circuit, so we wish to optimize the circuit complexity of the *transition function* of the ISA: the size of the smallest circuit that, given two adjacent states in the trace, verifies that the transition between the two indeed respects the ISA specification.[1]

We thus seek an ISA that strikes a balance between two opposing requirements: (1) the need for a transition function of small circuit complexity and (2) the need to produce small and fast machine code, in particular when compiling from high-level programming languages. Rich architectures allow for smaller code and shorter execution trace but have transition functions of higher circuit complexity, while minimalistic architectures require longer machine code and longer execution traces, but enjoy transition functions with smaller circuit complexity.

Modern ISAs designed for general purpose CPUs (such as x86) are complex instruction set computer (CISC) machines: they support many elaborate instructions (e.g., a round of AES [Gue12]) and addressing modes. Less rich ISAs are reduced instruction set computer (RISC) machines designed for devices like smartphones (ARM) and embedded microcontrollers (Atmel AVR).

In sum, we seek a minimal ISA that enables us to design a transition function with small circuit complexity, and yet allows reasonable overheads in code size and execution time (relative to richer ISAs).

A Custom ISA. In light of the above, we designed an instruction set architecture, named TinyRAM, that is tailored for our setting. TinyRAM is a minimalistic RISC random-access machine with a Harvard architecture and word-addressable random-access memory. It has two parameters: the *word size*, denoted W, and the *number of registers*, denoted K. The global state of the machine at any time consists of:

- the *program counter*, denoted pc; it consists of W bits;
- K general-purpose *registers*, denoted $\mathtt{r}0, \mathtt{r}1, \ldots, \mathtt{r}(K-1)$, each with of W bits;
- the *(condition) flag*, denoted flag; it consists of a single bit; and
- *memory*, which is a linear array of 2^W words of W bits each.

In addition, the machine has two *input tapes*, each containing a string of W-bit words. Each tape can be read sequentially in one direction only. The first input tape is for the *primary input*, denoted x; the second input tape is for the *auxiliary input*, denoted w. We treat the primary input as given, and the auxiliary input as nondeterministic advice. (See Definition 1 below.)

We carefully selected the instructions of TinyRAM so to support relatively efficient compilation from high-level programming languages (like C), as discussed in Section 2.2, and, furthermore, allow for small circuits implementing its transition function (and other checks), as discussed in Section 2.3. Briefly, the instruction set of TinyRAM includes simple load and store instructions for accessing random-access memory, as well as simple integer, shift, logical, compare, move, and jump instructions. TinyRAM can efficiently implement complex control flow, loops, subroutines, recursion, and so on. Complicated instructions, such as floating-point arithmetic, are not directly supported and can be implemented "in software" by TinyRAM programs. Supporting only

[1] This does not include the task of checking the correctness of values loaded from random-access memory. Memory consistency is efficiently handled separately. See Section 2.3.

fairly simple load and store operations is important for efficiently verifying consistency of random-access memory; see Section 2.3.

In keeping with the setting of verifiying computation, the only input to TinyRAM programs is via its two input tapes, and the only output is via an `accept` instruction, which also terminates execution.[2]

So far we have only informally discussed "correctness of TinyRAM program execution". This notion is formalized by defining a TinyRAM universal language.

Definition 1. *Fix the word size W and number of registers K. Let $\mathbf{P}$ be a* TinyRAM *program, let x and w be strings of W-bit words. We say that $\mathbf{P}(x, w)$* **accepts in** T **steps** *if $\mathbf{P}$, with x on its first input tape and w on the second, executes the instruction* `accept` *in step T.*
The TinyRAM **universal language** *consists of the triples $(\mathbf{P}, x, T)$ where $\mathbf{P}$ is a* TinyRAM *program, x is a string of W-bit words, and T is a time bound, such that there exists a string w of W-bit words for which $\mathbf{P}(x, w)$ accepts in T steps.*

A specification for the TinyRAM architecture can be found in [BCG+13].

2.2 A Compiler from C to TinyRAM

The GCC compiler is a versatile framework supporting many source languages (e.g., C and Java) and many target languages (e.g., x86 and ARM assembly). Internally, the GCC compiler is partitioned into two main modules [StGDC13]. The *frontend* is responsible for converting a program written in a high-level programming language like C or Java into an intermediate representation language called *Register Transfer Language* (RTL). The *backend* is responsible for optimizing and converting RTL code into corresponding assembly code for a given architecture.

In order to automatically generate TinyRAM assembly for problems of interest, we have implemented a prototype of a GCC backend for converting RTL code to TinyRAM assembly code. Our prototype backend works with the C frontend, and can be extended to other programming languages by combining it with suitable GCC frontends (and providing the requisite standard libraries). Concretely, we have a prototype that can compile a subset of C to TinyRAM, with word size $W \in \{8, 16\}$ and number of registers $K \geq 15$.

Because TinyRAM's instruction set is quite minimal, any operation not directly supported by TinyRAM "hardware" needs to be implemented in "software". This incurs overheads in both the *code size* (the number of lines in an assembly code) and *execution time* (the number of machine steps required to execute a piece of code). By running experiments, we established that both of these overheads are not large, as discussed next.

Code Size Overhead. We first evaluate the code size produced when compiling C code examplesinto TinyRAM assembly using our GCC port, compared to the code produced by standard GCC for some common architectures: x86, ARM and AVR. (We used the $-$O1 optimization flag in all cases.) Our results show that, compared to the RISC architectures (ARM and AVR), the resulting TinyRAM code is at most twice larger than

[2] For ease of development, the TinyRAM simulator also supports debugging instructions that produce additional outputs. These are excluded from the execution trace and not verified.

ARM and significantly smaller than AVR. Compared to x86, which is a very rich CISC architecture, TinyRAM code is up to three times bigger. We deduce that, at least for the program styles represented by these examples, the TinyRAM architecture allows for compilation into compact programs. See the extended version of this paper for details.

Execution Time Overhead. The circuits ultimately produced by our reduction have size $O(T \log T)$, where T is the execution time (measured in machine steps). This execution time depends on the choice of architecture, and we wish to ensure that TinyRAM does not necessitate very long execution times due to deficiencies in the instruction set.

To evaluate this, we compiled examples of C code into both TinyRAM machine code and x86 machine code. Our results show that in terms of execution time measured in machine steps (i.e., clock cycles), TinyRAM is slower than x86 by a factor of merely 2 to 6, for examples that represent some realistic computations. This is despite x86 being a very rich CISC architecture that is heavily optimized for minimizing clock cycles, which is typically implemented using many millions of gates. (Recall the difference of executing vs. verifying, discussed in Section 2.1.) See the extended version for details.

These small overheads are more than compensated by the fact that TinyRAM has a very compact transition function circuit. For instance, for a wordsize $W = 16$ and number of registers $K = 16$, and for a program with 100 instructions, the transition function consists of only 708 gates.

In summary, our experiments show that, even when working with a minimalistic architecture such as TinyRAM, we do not incur large overheads in code size or execution time. In Section 2.3, we discuss the circuit complexity of TinyRAM's transition function and how to efficiently verify TinyRAM traces.

2.3 An Efficient Reduction from TinyRAM to Circuit Satisfiability

The following describes our efficient reduction from correctness of TinyRAM executions to $\mathbb{F}$-arithmetic circuit satisfiability, for any prime field $\mathbb{F}$ of sufficiently large size.

The Reduction Notion. In our setting, a *(circuit) reduction* is a triple of functions $(\mathsf{circ}, \mathsf{wit}, \mathsf{wit}^{-1})$ working as follows. The *circuit generator* function, circ, maps a TinyRAM program $\mathbf{P}$, time bound T, and primary input size n to a corresponding $\mathbb{F}$-arithmetic circuit C that encodes the correct computation of $\mathbf{P}$ for at most T steps on primary inputs of n words. The *witness map* function, wit, maps a pair of primary and auxiliary inputs (x, w) that make $\mathbf{P}$ accept in T steps, to a satisfying assignment a for $C(x, \cdot)$. The *inverse witness map* function, wit^{-1}, maps a satisfying assignment a for $C(x, \cdot)$ to w with the property that (x, w) makes $\mathbf{P}$ accept in T steps.

Definition 2. *A* **reduction** *from* TinyRAM *(for a word size W and number of registers K) to $\mathbb{F}$-arithmetic circuit satisfiability is a triple of functions* $(\mathsf{circ}, \mathsf{wit}, \mathsf{wit}^{-1})$ *such that, for every* TinyRAM *program* $\mathbf{P}$*, time bound T, and primary input size n, the following hold:*

- $C := \mathsf{circ}(\mathbf{P}, T, n)$ *is an $\mathbb{F}$-arithmetic circuit from $\mathbb{F}^{W \cdot n} \times \mathbb{F}^h$ to $\mathbb{F}^\ell$ for some h, ℓ; C's gates are bilinear;*[3]

[3] A gate with inputs $x_1, \ldots, x_n$ is *bilinear* if the output is $\langle \mathbf{a}, (1, x_1, \ldots, x_n) \rangle \cdot \langle \mathbf{b}, (1, x_1, \ldots, x_n) \rangle$ for some $\mathbf{a}, \mathbf{b} \in \mathbb{F}^{n+1}$.

- *for every* (x, w) *such that* $\mathbf{P}(x, w)$ *accepts in* T *steps,* $C\big(x, \mathsf{wit}(\mathbf{P}, T, x, w)\big) = 0^\ell$*;*
- *for every* (x, a) *such that* $C(x, a) = 0^\ell$*,* $\mathbf{P}\big(x, \mathsf{wit}^{-1}(\mathbf{P}, T, x, a)\big)$ *accepts in* T *steps.*

The work on fast reductions of Ben-Sasson et al. [BCGT13a] implies a reduction $(\mathsf{circ}, \mathsf{wit}, \mathsf{wit}^{-1})$ where $|C| := O(T(\log T)^2)$ and $\mathsf{circ}, \mathsf{wit}, \mathsf{wit}^{-1}$ all run in $O(T(\log T)^2)$ time.[4] In our work, we optimize and implement a reduction that builds on the theoretical approach of [BCGT13a]. We shall focus our attention only on the efficiency of the circuit and witness maps (i.e., circ and wit), because these actually need to be run in practice. Before discussing our work, however, we briefly review the approach of [BCGT13a].

The Reduction in [BCGT13a]. We begin with necessary basic definitions.

- A *(local) state* of TinyRAM, denoted S, is a string of $(W + KW + 1)$ bits, encoding the values of the program counter, K registers, and condition flag at a given step.
- The *transition function* of TinyRAM, denoted Π_{TF}, is the predicate that, given a TinyRAM program $\mathbf{P}$ and two states S and S', outputs 1 if and only if the machine in state S can transition (for *some* choice of values in random-access memory) to the state S' in the next step, according to the program $\mathbf{P}$.[5]
- An *execution trace*[6] for a TinyRAM program $\mathbf{P}$, time bound T, and primary input x is a sequence of states $\mathsf{tr} = (S_1, \ldots, S_T)$. An execution trace tr is *valid* if there exists an auxiliary input w such that the sequence of states induced by $\mathbf{P}$ running with input tapes (x, w) is tr.

The goal is to design an $\mathbb{F}$-arithmetic circuit C for verifying that tr is valid that is as small as possible. This is done in three steps, as follows.

Step 1: Code Consistency. Let C_{TF} be a circuit that implements the transition function Π_{TF} of TinyRAM: namely, $C_{\mathsf{TF}}(\mathbf{P}, S, S') = 1$ if and only if $\Pi_{\mathsf{TF}}(\mathbf{P}, S, S') = 1$. By invoking C_{TF} on each pair of successive states of tr, we can verify every state transition in the trace tr, i.e., ensure that $\Pi_{\mathsf{TF}}(\mathbf{P}, S_i, S_{i+1}) = 1$ for $i = 1, \ldots, T - 1$. Doing so gives rise to a sub-circuit of C, consisting of T copies of C_{TF}, that, when given as input tr, checks that tr is *code-consistent*.

Step 2: Memory Consistency. The global state of a random-access machine, however, also includes memory. In particular, in order to verify that tr is valid, we *also* need to verify that tr is *memory-consistent*: namely, that every load operation from an address in memory actually retrieves the value of the last store to that address.

But the accesses to memory of a program $\mathbf{P}$ depend on the inputs x and w. Hence, in general, at each time step i any of the addresses in memory could be accessed by the program. The naive solution of designing the verification circuit C to maintain a

[4] Given a space bound S on the computation of $\mathbf{P}$ on (x, w), Ben-Sasson et al. also present a reduction where $|C|$ is only $O(T \log T \log S)$. We have so far not considered this additional, significantly more complex, optimization.

[5] Traditionally, the transition function is the function that, given the global state of a machine as input, outputs the next state. We abuse this terminology, and use it for the function that, given two local states S, S', decides whether the second can follow the first.

[6] An execution trace is also at times known as a *computation transcript* [BCGT13a].

snapshot of memory for each time step is *not* efficient: such a circuit has size that is $\Omega(T^2)$. (All previous circuit generators either adopt the naive solution or restrict a program's memory accesses to be known at compile time.)

Ben-Sasson et al. [BCGT13a] do *not* adopt the naive solution (or restrict a program's memory accesses), but instead take an approach that is more efficient; the approach builds on classical results on quasilinear-time nondeterministic reductions [Sch78, GS89, Rob91]. The high-level idea in [BCGT13a] is that memory consistency would be easier to verify if the circuit C were to also have, as additional input, the *same* trace tr but sorted according to accessed memory addresses (and breaking ties via timestamps); let us denote this sorted trace by $\mathsf{MemSort}(\mathsf{tr})$. Concretely, one can define another "local" predicate Π_{MC} such that, if Π_{MC} is satisfied by each pair of adjacent states in $\mathsf{MemSort}(\mathsf{tr})$ (and, in addition, tr is code-consistent) then tr is valid. We can then augment C with T copies of a sub-circuit C_{MC} that verifies the predicate Π_{MC} on $\mathsf{MemSort}(\mathsf{tr})$. The circuit C is thus left to verify that the auxiliary input $\mathsf{MemSort}(\mathsf{tr})$ is the result of sorting tr.

Step 3: Routing Network. The circuit C can efficiently perform this check if it is given yet another additional input: (alleged) routing decisions for a routing network which permutes tr into $\mathsf{MemSort}(\mathsf{tr})$. A T-packet *routing network* is a directed graph with T sources, T sinks, and inner nodes (switches) such that, for any permutation $\pi\colon [T] \to [T]$, there are routing decisions for the switches that cause T packets at the sources to travel to the T sinks, according to the permutation π, and without using a switch twice (i.e., with no congestion). One such a network is the Beneš network [Ben65], which has $O(\log T)$ layers of T nodes each, and each node in a layer is connected to two nodes in the next layer. The idea is to interpret the switch settings in a routing network as a coloring on the routing network. Crucially, verifying that the given switch settings (i.e., a coloring of the network) implement *some* permutation from the input nodes to the output nodes can be done via simple and local routing constraints; furthermore, given that the switches implement some permutation, verifying that they implement the sorting permutation is easy to verify too. Overall we obtain a certain graph-coloring problem all of whose constraints can be evaluated by a circuit of size $T \cdot O((\log T)^2)$, which we add to C.

In Sum. The approach from [BCGT13a] described in the above paragraphs yields a circuit C of size $T \cdot \big(|C_{\mathsf{TF}}| + |C_{\mathsf{MC}}| + O((\log T)^2)\big)$ for verifying a T-step trace.

Our Optimized Reduction. As mentioned, in our work we optimize and implement the theoretical approach of Ben-Sasson et al. [BCGT13a]. Despite the excellent asymptotic efficiency of the approach, getting to the point in which the verification circuit C has a manageable size in practice proved quite challenging, both theoretically and programmatically. For instance: while (as discussed in Section 2.1) we devised TinyRAM to facilitate the design of a small circuit C_{TF} for the transition function Π_{TF}, how small of a circuit can we actually design? And how well does its size scale with, say, the word size W, number of registers K, and program size $|\mathbf{P}|$?

Our Circuit Generator. At high level, our main technical contribution is leveraging
(1) *"native" arithmetic* in the field $\mathbb{F}$ (which for us is a prime field) and
(2) *nondeterministic advice*

so to achieve highly-optimized implementations of C_{TF}, C_{MC}, and routing constraints, and ultimately obtain drastic improvements in the size of the verification circuit C output by our circuit generator circ.

To illustrate the use of (1) and (2), consider the basic task of *multiplexing bit vectors*, used numerous times in C. Given n vectors $\mathbf{a}_1, \ldots, \mathbf{a}_n$ of ℓ bits each and a $\lceil \log n \rceil$-bit index $\mathbf{i}$, we seek a small $\mathbb{F}$-arithmetic circuit that computes the vector selected by the index. The naive multiplexer circuit requires $O(n(\ell + \log n))$ bilinear gates. In contrast, by relying on (1) and (2), we design a multiplexer circuit that needs only $O(n\lceil \frac{\ell}{|\mathbb{F}|} \rceil)$ bilinear gates. The efficiency improvement is significant because we ultimately need to work with cryptographically-large fields; for instance, in our setting, if $n = \ell = 16$, the naive implementation uses 320 gates while we only use 51.

The idea of our multiplexer construction is as follows. Suppose, first, that every input vector, as well as the index, were represented as integers, and we only had to design a $\mathbb{Z}$-arithmetic circuit to output the integer representing the selected bit vector. In this case, we could easily construct a nondeterministic $\mathbb{Z}$-arithmetic circuit of size $O(n)$ (with bilinear gates of unbounded fan-in). However, the vectors are only given to us as strings of bits, and we need to work with $\mathbb{F}$-arithmetic circuits. This gap motivates two fundamental operations: *packing* and *unpacking* of bit vectors. Packing denotes mapping a bit vector to a sequence of field elements efficiently storing these bits, and unpacking denotes the inverse operation. The packing operation is very efficient: in the prime field $\mathbb{F}_p$ with $p \geq 2^\ell$, a single gate suffices to compute $\sum_{i=1}^{\ell} 2^{i-1} a_i$ from the input $a_1, \ldots, a_\ell$. The inverse operation is much more expensive to compute directly, but we can nondeterministically *guess* the answer and verify it using a single gate. In general, $p \geq 2^\ell$ need not hold, so we use $\lceil \frac{\ell}{|\mathbb{F}|} \rceil$ field elements to store an ℓ-bit vector. Given the aforementioned efficient packing operations, our multiplexer construction works as follows: it guesses the selected ℓ-bit vector, then computes the integers corresponding to the input ℓ-bit vectors as well as the index, and then verifies the guess by selecting the correct integer according to the (integer) index.

More generally, we have found that, throughout our circuit generator, it is often advantageous to maintain, alongside certain vectors $\mathbf{a}$, also the corresponding integer $\sum_i 2^{i-1} a_i$. We believe that packing and unpacking operations will be crucial for drastically decreasing the size of circuits used in future circuit generators.

With these techniques in mind, we proceed to describe the circuit generator.

- *Designing the transition function circuit* C_{TF}. The circuit C_{TF} is the most complex sub-circuit of C. The size of C_{TF} is dominated by the size of sub-circuits for *multiplexing bit strings* (for instruction fetch, register fetch, and so on) and of the *arithmetic logic unit* (ALU), which executes the architecture's non-memory operations.
  To obtain an efficient implementation of the ALU, we again make use of field arithmetic and nondeterministic advice. Since we work over a prime field of large characteristic, field arithmetic looks like integer arithmetic whenever there is no "wrap around". Thus, after fetching the arguments to an operation, we derive from each argument's binary representation also the corresponding integer. Then, each operation in the ALU computes on the integer representation, instead of the binary one, when it is more efficient to do so. For instance, we use this idea to compute result and overflow information for addition, subtraction, and multiplication with only $2W$,

Table 1. Number of gates in C_{TF} as a function of $|\mathbf{P}|, W, K$

$\lvert\mathbf{P}\rvert = 10/100/1000$	$W = 8$	$W = 16$	$W = 32$
$K = 8$	416 / 506 / 1406	520 / 610 / 1510	728 / 818 / 1718
$K = 16$	514 / 604 / 1504	618 / 708 / 1608	826 / 916 / 1816
$K = 32$	708 / 798 / 1698	812 / 902 / 1802	1020 / 1110 / 2010

$2W$, and $3W$ bilinear gates, respectively; as for division, we guess the result and verify it with a multiplication. In each case, the integer output by an operation can be "unpacked" into its binary representation, via nondeterministic advice. By carefully implementing each operation, we obtain an ALU that, e.g., when $W = 16$ only has 296 gates.

Given efficient implementations of multiplexing and ALU, it is not difficult to obtain an efficient implementation of C_{TF}. Table 1 shows the number of gates in our implementation of C_{TF} for $|\mathbf{P}| \in \{10, 10^2, 10^3\}$, $W \in \{8, 16, 32\}$ and $K \in \{8, 16, 32\}$.

- *Designing the memory consistency circuit* C_{MC}. The predicate Π_{MC} is not as complex as the transition function Π_{TF}, but it is still important to design a small circuit C_{MC} for it. The bottleneck in the computation of Π_{MC} is again multiplexing, this time for fetching the two arguments of a memory operation. Thus, the natural approach here would be to use additional copies of our efficient multiplexer circuit. Instead, we show how to avoid additional multiplexing altogether by "stealing" certain intermediate computations from C_{TF}. We thereby obtain a circuit for C_{MC} that only contains two integer comparisons and few other logical operations. For instance, when $W = 16$, C_{MC} only costs us 60 additional gates.
- *Checking routing constraints.* Asymptotically, the routing constraints on the routing network are the most expensive sub-circuit of C. It is thus crucial to compute these constraints as efficiently as possible. A first concern is to minimize the size of a packet routed through the network. Instead of setting a packet to be a local state of the machine, which consists of $(W + KW + 1)$ bits, we show that it only suffices to send a much smaller packet, consisting of about $2W$ bits, obtained from intermediate computations of C_{TF}. This optimization in fact leads to another important one: now that a packet is as small as only about $2W$ bits, we can "pack" all the bits on a *single* field element (in our setting, $\mathbb{F}$ has size at least 2^W); then, because the packets consist of single field elements, computing the routing constraints becomes particularly simple: only one bilinear gate per vertex. Concretely, the gate at a given vertex checks whether the vertex's packet is equal to at least one of the packets at the two neighbor vertices in the next layer. Overall, when T is a power of 2, all the routing constraints can be verified with only $2 \cdot T \cdot \log T$ gates. (We thus also obtain an asymptotic improvement, by a $\log T$ factor, over the circuit size in [BCGT13a], where routing constraints required $O(T(\log T)^2)$ gates.)

Of course, there are numerous additional details that go into our final construction of the verification circuit C. Overall, say that for concreteness we fix $W = 16$, $K = 16$, and $|\mathbf{P}| = 100$, then we get

$$|C| = A \cdot T \cdot \log T + B \cdot T + C \ \text{, where } A = 4,\ B = 1116 \text{ and } C = 307.$$

In particular, for $\log T < 20$, every cycle of $\mathsf{TinyRAM}$ computation costs ≈ 1200 gates. Note that, while the gate count per cycle increases as T increases (as the number of routing constraints grows as $O(T \log T)$), the growth rate is slow: doubling T costs only $4 + o(1)$ additional gates per cycle.

Our Witness Map. Thus far, we have focused on achieving *soundness*: verifying the validity of an execution trace of a $\mathsf{TinyRAM}$ program $\mathbf{P}$ by using the circuit $C := \mathsf{circ}(\mathbf{P}, T, n)$ output by the circuit generator circ. The circuit generator is run by the key generator when computing the public parameters. For *completeness*, we need to implement a witness map $\mathsf{wit}(\mathbf{P}, T, x, w)$ that computes a satisfying assignment a for $C(x, \cdot)$, whenever $\mathbf{P}(x, w)$ accepts in T steps. The witness map is executed by the prover when generating a proof. See the extended version for details on this map.

3 Verifying Circuit Satisfiability via Linear PCPs

As summarized in Section 1.3, we have implemented a zk-SNARK for circuit satisfiability; see Section 1.1 for an informal definition of this cryptographic primitive, or the extended version of this paper for a formal one. In this section we describe the design and performance of this part of our system.

3.1 A Transformation from Any Linear PCP

We begin by discussing efficiency aspects of the transformation from a linear PCP to a corresponding SNARK. To do so, we first recall (at high level) the transformation itself.

Constructing a SNARK from a linear PCP. The transformation of Bitansky et al. [BCI+13] consists of an information-theoretic step followed by cryptographic step.

- *Step 1 (information-theoretic):* compile the linear PCP into a 2-message *linear interactive proof* (linear IP), i.e., one where the prover is restricted to only apply linear functions to the verifier's message.
  This is achieved by adding a *consistency-check query*, which is a random linear combination of the linear PCP queries. In more detail, if the linear PCP has k queries each with m elements from a field $\mathbb{F}$, in the resulting linear IP the verifier sends to the prover a single message $\boldsymbol{q}$ consisting of $m' = (k+1)m$ elements in $\mathbb{F}$; the message $\boldsymbol{q}$ is the concatenation of the k linear PCP queries and the consistency-check query. A (potentially malicious) prover is restricted to only apply linear functions to $\boldsymbol{q}$, i.e., reply with a vector $\boldsymbol{a}^* \in \mathbb{F}^{k+1}$ such that $\boldsymbol{a}^* = \Pi^* \boldsymbol{q} + \boldsymbol{b}^*$ for some $\Pi^* \in \mathbb{F}^{(k+1) \times m'}$ and $\boldsymbol{b}^* \in \mathbb{F}^{k+1}$. The honest prover simply returns the vector $\boldsymbol{a} = (a_1, \dots, a_{k+1})$ where $a_i = \langle \boldsymbol{\pi}, \boldsymbol{q}_i \rangle$, $\boldsymbol{q}_i$ is the i-th m-element block of $\boldsymbol{q}$, and $\boldsymbol{\pi}$ is the linear PCP. A prover's message $\boldsymbol{a}^*$ is verified by checking consistency of a^*_{k+1} with $a^*_1, \dots, a^*_k$ and then invoking the linear PCP decision predicate on $a^*_1, \dots, a^*_k$; the consistency check ensures that $a^*_i = \langle \boldsymbol{\pi}^*, \boldsymbol{q}_i \rangle$ for *some* linear PCP $\boldsymbol{\pi}^*$.
- *Step 2 (cryptographic):* compile the linear IP into a SNARK, by forcing any polynomial-size malicious prover to act as if it were a linear function.
  This is achieved using a cryptographic encoding $\mathsf{Enc}(\cdot)$ with the following properties.
  (i) It allows public testing of quadratic predicates on encoded elements.

(ii) It provides a certain notion of one-way security to encoded elements.

(iii) It ensures that any polynomial-size prover can only perform linear operations on the encoded elements, “up to” information leaked by the encoding.[7]

Given $\mathsf{Enc}(\cdot)$, the compilation is then conceptually simple. The SNARK generator $G(1^\lambda, C)$ samples a verifier message $\boldsymbol{q} \in \mathbb{F}^{m'}$ (which depends on the circuit C but not its input) for the linear IP, and outputs, as a proving key, the encoding $\mathsf{Enc}(\boldsymbol{q}) = (\mathsf{Enc}(q_i))_{i=1}^{m'}$. (We omit here the discussion of how the short verification key is generated.) Starting from $\mathsf{Enc}(\boldsymbol{q})$ and a linear PCP $\boldsymbol{\pi}$, the honest SNARK prover P homomorphically evaluates the inner products $\langle \boldsymbol{\pi}, \boldsymbol{q}_i \rangle$ and returns as a proof the resulting encoded answers. The SNARK verifier checks a proof by running the linear IP decision predicate on the encoded answers.

For precise definitions and details, see [BCI+13].

Computational Overheads. The transformation from a linear PCP to a SNARK introduces several computational overheads. In Step 1, the only overhead is due to the consistency-check query, and is minor. However, the cryptographic overheads in Step 2 are significant, and require optimizations for practical use.

Specifically, after sampling $\boldsymbol{q} \in \mathbb{F}^{m'}$, the SNARK generator G must compute $\mathsf{Enc}(\boldsymbol{q}) = (\mathsf{Enc}(q_i))_{i=1}^{m'}$. In other words G needs to compute the encoding of m' field elements, where m' is on the order of the size of the circuit C. Furthermore, after computing a linear proof oracle $\boldsymbol{\pi} \in \mathbb{F}^m$, the honest SNARK prover P needs to homomorphically evaluate, for $i = 1, \ldots, k+1$, the inner product $\langle \boldsymbol{\pi}, \boldsymbol{q}_i \rangle$ to obtain $\mathsf{Enc}(\langle \boldsymbol{\pi}, \boldsymbol{q}_i \rangle)$.

In our case, the encoding is $\mathsf{Enc}(\gamma) = (g^\gamma, h^\gamma)$ where $g \in G_1, h \in G_2$ and G_1, G_2 are prime-order groups; the linear homomorphism is $\mathsf{Enc}(a\gamma + b\delta) = \mathsf{Enc}(\gamma)^a \mathsf{Enc}(\delta)^b$ with coordinate-wise multiplication and exponentiation. Therefore, both G and P need to compute a large number of cryptographic exponentiations. These operations greatly affect the complexity of G and P, and must be performed efficiently.

Efficiency Optimizations. We address the cryptographic bottleneck by using multi-exponentiation algorithms and parallelization. See the extended version of this paper for the impact of these optimizations.

3.2 An Efficient Linear PCP

In the previous section we discussed how to ensure that the transformation from a linear PCP to a SNARK adds as little computational overhead as possible. In this section, we discuss the problem of implementing a linear PCP (to give as input to the transformation) that is as efficient as possible.

Our Linear PCP. Our starting point is the work on *quadratic-span programs* (QSPs) and *quadratic-arithmetic programs* (QAPs) of Gennaro et al. [GGPR13]. Indeed, Bitansky et al. [BCI+13] observed that any QSP for a relation $\mathcal{R}$ yields a corresponding 3-query linear PCP for $\mathcal{R}$, and any QAP for a relation $\mathcal{R}$ yields a corresponding 4-query linear PCP for $\mathcal{R}$. By following the QAP approach of [GGPR13], we design a

[7] Since the encoding cannot provide semantic security (due to the functionality requirement of allowing for evaluation of quadratic predicates on encoded elements) but only a notion of one-way security, a limited amount of information is necessarily leaked.

linear PCP that trades an increased number of 5 queries for a linear PCP that, while keeping essentially optimal asymptotics, enjoys excellent efficiency in practice.

Concretely, for checking membership in the language $\mathcal{L}_C$ for a circuit C, our linear PCP has only 5 queries of $2|C|$ field elements each (and sampling the 5 queries needs only a single random field element); generating the queries can be done in linear time. The 5 answers of the queries can be verified via 2 quadratic polynomials using only $2n + 9$ field operations, where n is the input size. The soundness error is $2|C|/|\mathbb{F}|$. Through a suitable use of FFTs, the honest prover can compute the linear proof oracle via an arithmetic circuit of size $O(|C| \log |C|)$ and depth $O(\log |C|)$ only. (In particular, the prover is highly parallelizable.)

Efficiency Optimizations. In practice, tailored FFT algorithms are more efficient than "generic" ones (i.e., ones that work over any finite field). To leverage the efficiency of tailored FFT algorithms, we further specialize our choice of elliptic curve so to ensure that G_1, G_2 are groups of a prime order p with $p - 1 = 2^\ell h$ for a large integer ℓ. This means that, in our linear PCP, we can choose the finite field $\mathbb{F} = \mathbb{F}_p$. In such a field, there is a primitive 2^ℓ-th root of unity, and multi-point evaluation/interpolation over domains consisting of roots of unity (or their multiplicative cosets) can be performed via very simple and efficient FFT algorithms. Furthermore, the choice $\mathbb{F} = \mathbb{F}_p$ also simplifies the linear-time algorithm for sampling queries.

Zero Knowledge. The transformation from a linear PCP to a SNARK is such that if the linear PCP is honest-verifier zero-knowledge (HVZK) then the SNARK is zero knowledge. (See the extended version of this paper for a definition of HVZK.) Thus, we need to ensure that our linear PCP is HVZK. Bitansky et al. [BCI$^+$13] showed a general transformation from a linear PCP to a HVZK linear PCP of similar efficiency. We do not rely on their general transformation. Instead, our linear PCP can be made HVZK with essentially no computational overhead, via a simple modification analogous to the one used in [GGPR13] to achieve zero knowledge. With this modification, we ensure that the SNARK obtained from our linear PCP has (statistical) zero knowledge.

For more details on our linear PCP construction, see the extended version of this paper.

3.3 Performance

Plugging our linear PCP for arithmetic circuits (Section 3.2) into the transformation (Section 3.1), we thus obtain an implementation of zk-SNARKs for arithmetic circuit satisfiability with excellent asymptotic efficiency: linear-time key generator, quasilinear-time prover, and linear-time verifier. Next, we discuss concrete performance.

Our algebraic setup is as follows: we work over $E(\mathbb{F}_q)$ where E is the elliptic curve $y^2 = x^3 + x$ and q is a prime of 512 bits; the order of the group is divisible by $p = 2^{159} + 2^{107} + 1$. This curve gives 128 bits of security. Our experiments are run on a machine with eight 2.4 GHz AMD Opteron 8431 6-core processors and 16 GB of RAM.

Performance of Key Generation. Given an arithmetic circuit $C \colon \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}$ as input, the SNARK key generator G outputs: a proving key σ of $(12|C| + 2n + 40)$ group elements and a verification key τ of $(n + 8)$ group elements. Each group element (when compressed) is 65 bytes. Only 8 random field elements need to be sampled for this computation. A small set of *public parameters* provides information, to both the prover and verifier, about the choice of elliptic curve; storing these public parameters only requires 310 bytes. The extended version of this paper includes performance graphs of $G(C)$ as a function of $|C|$. For instance, when $|C| \approx 2 \cdot 10^6$, G performs $\approx 4.2 \cdot 10^9$ field operations in less than 20 minutes.

Performance of Proving. Given σ and (x, a) in the relation $\mathcal{R}_C$, the SNARK prover outputs a proof consisting of 12 group elements. As before, each group element (when compressed) is 65 bytes, so the proof length in bytes is 780. The extended version of this paper includes graphs of $P(\sigma, x, a)$ as a function of $|C|$. For instance, when $|C| \approx 2 \cdot 10^6$, P performs $\approx 3.3 \cdot 10^9$ field operations in less than 15 minutes.

Performance of verifying. Given τ, an input x, and a proof π, the SNARK verifier computes the decision bit. To do so, the verifier evaluates 21 pairings and solves a multi-exponentiation problem of size $|x|$.The extended version of the paper includes performance graphs of $V(\tau, x, \pi)$ as a function of $|x|$. For instance:

- when $|x| \leq 2^6$, V performs $\approx 2.2 \cdot 10^5$ field operations in less than 50 milliseconds;
- when $|x| \leq 2^{17}$, V performs $\approx 1.3 \cdot 10^7$ field operations in less than 20 seconds.

We emphasize that the above performance holds *no matter how large is the circuit C.*

Acknowledgments. The authors gratefully thank the members of the programming team: Ohad Barta, Lior Greenblatt, Shaul Kfir, Michael Riabzev, Gil Timnat, and Arnon Yogev. We also thank Lev Pachmanov for helping out with TinyRAM programming; Dan Berstein, Tanja Lange, Peter Schwabe, and Andrew Sutherland for discussions about elliptic curves; and Ron Rivest and Nickolai Zeldovich for helpful comments and discussions. We thank Nickolai Zeldovich for the use of his group's compute nodes.

The research leading to these results — in particular, funding of the aforementioned programming team — has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 240258. This work was partially supported by the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370; by the Check Point Institute for Information Security; by the Israeli Ministry of Science and Technology, and by the Israeli Centers of Research Excellence I-CORE program (center 4/11).

References

[ABB$^+$12] Almeida, J.B., Barbosa, M., Bangerter, E., Barthe, G., Krenn, S., Béguelin, S.Z.: Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols. In: CCS 2012 (2012)

[BBK$^+$09] Bangerter, E., Barzan, S., Krenn, S., Sadeghi, A.-R., Schneider, T.: Bringing zero-knowledge proofs of knowledge to practice (2009)

[BCC88] Brassard, G., Chaum, D., Crépeau, C.: Minimum disclosure proofs of knowledge. Journal of Computer and System Sciences 37(2), 156–189 (1988)

[BCC$^+$09] Belenkiy, M., Camenisch, J., Chase, M., Kohlweiss, M., Lysyanskaya, A., Shacham, H.: Randomizable proofs and delegatable anonymous credentials. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 108–125. Springer, Heidelberg (2009)

[BCCT13] Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: Recursive composition and bootstrapping for SNARKs and proof-carrying data. In: STOC 2013 (2013)

[BCG$^+$13] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: TinyRAM architecture specification v1.00 (2013), `http://scipr-lab.org/tinyram`

[BCGT13a] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In: ITCS (2013)

[BCGT13b] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: On the concrete efficiency of probabilistically-checkable proofs. In: STOC 2013 (2013)

[BCI$^+$13] Bitansky, N., Chiesa, A., Ishai, Y., Paneth, O., Ostrovsky, R.: Succinct non-interactive arguments via linear interactive proofs. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 315–333. Springer, Heidelberg (2013)

[BCKL08] Belenkiy, M., Chase, M., Kohlweiss, M., Lysyanskaya, A.: P-signatures and noninteractive anonymous credentials. In: Canetti, R. (ed.) TCC 2008. LNCS, vol. 4948, pp. 356–374. Springer, Heidelberg (2008)

[BDNP08] Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: CCS 2008 (2008)

[Ben65] Beneš, V.E.: Mathematical theory of connecting networks and telephone traffic. Academic Press, New York (1965)

[Ber02] Bernstein, D.J.: Pippenger's exponentiation algorithm (2002), `http://cr.yp.to/papers/pippenger.pdf`

[BFLS91] Babai, L., Fortnow, L., Levin, L.A., Szegedy, M.: Checking computations in polylogarithmic time. In: STOC 1991 (1991)

[BGW88] Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: STOC 1988 (1988)

[BHZ87] Boppana, R.B., Håstad, J., Zachos, S.: Does co-NP have short interactive proofs? Information Processing Letters 25(2), 127–132 (1987)

[BP04] Bellare, M., Palacio, A.: The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 273–289. Springer, Heidelberg (2004)

[BS08] Ben-Sasson, E., Sudan, M.: Short PCPs with polylog query complexity. SIAM Journal on Computing 38(2) (2008)

[BW06] Boyen, X., Waters, B.: Compact group signatures without random oracles. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 427–444. Springer, Heidelberg (2006)

[CKLM12] Chase, M., Kohlweiss, M., Lysyanskaya, A., Meiklejohn, S.: Malleable proof systems and applications. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 281–300. Springer, Heidelberg (2012)

[Dam92] Damgård, I.: Towards practical public key systems secure against chosen ciphertext attacks. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 445–456. Springer, Heidelberg (1992)

[Din07] Dinur, I.: The PCP theorem by gap amplification. Journal of the ACM 54(3) (2007)

[GGPR13] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 626–645. Springer, Heidelberg (2013)

[GH98] Goldreich, O., Håstad, J.: On the complexity of interactive proofs with bounded communication. Information Processing Letters 67(4), 205–214 (1998)

[GMW87] Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC 1987 (1987)

[Gro05] Groth, J.: Non-interactive zero-knowledge arguments for voting. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 467–482. Springer, Heidelberg (2005)

[Gro06] Groth, J.: Simulation-sound NIZK proofs for a practical language and constant size group signatures. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 444–459. Springer, Heidelberg (2006)

[Gro10] Groth, J.: Short non-interactive zero-knowledge proofs. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 341–358. Springer, Heidelberg (2010)

[GS89] Gurevich, Y., Shelah, S.: Nearly linear time. In: Logic at Botik 1989, Symposium on Logical Foundations of Computer Science, pp. 108–118 (1989)

[Gue12] Gueron, S.: Intel advanced encryption standard (AES) instructions set (February 2012)

[GVW02] Goldreich, O., Vadhan, S., Wigderson, A.: On interactive proofs with a laconic prover. Computational Complexity 11(1/2), 1–53 (2002)

[HT98] Hada, S., Tanaka, T.: On the existence of 3-round zero-knowledge protocols. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 408–423. Springer, Heidelberg (1998)

[KMO01] Katz, J., Myers, S., Ostrovsky, R.: Cryptographic counters and applications to electronic voting. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 78–92. Springer, Heidelberg (2001)

[Lip11] Lipmaa, H.: Two simple code-verification voting protocols. Cryptology ePrint Archive, Report 2011/317 (2011)

[Lip12] Lipmaa, H.: Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In: Cramer, R. (ed.) TCC 2012. LNCS, vol. 7194, pp. 169–189. Springer, Heidelberg (2012)

[Mic00] Micali, S.: Computationally sound proofs. SIAM Journal on Computing 30(4), 1253–1298 (2000); Preliminary version appeared in FOCS 1994 (1994)

[MNPS04] Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay — a secure two-party computation system. In: SSYM 2004 (2004)

[MR08] Moshkovitz, D., Raz, R.: Two-query PCP with subconstant error. Journal of the ACM 57, 1–29 (2008); Preliminary version appeared in FOCS 2008 (2008)

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

[Sch78] Schnorr, C.-P.: Satisfiability is quasilinear complete in NQL. Journal of the ACM 25, 136–145 (1978)

[StGDC13] Stallman, R.M., and the GCC Developer Community: GNU compiler collection internals (2013), `http://gcc.gnu.org/onlinedocs/gccint.pdf`

[Wee05] Wee, H.: On round-efficient argument systems. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 140–152. Springer, Heidelberg (2005)