Scalable Zero Knowledge via Cycles of Elliptic Curves

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

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

Abstract. Non-interactive zero-knowledge proofs of knowledge for general NP statements are a powerful cryptographic primitive, both in theory and in practical applications. Recently, much research has focused on achieving an additional property, *succinctness*, requiring the proof to be very short and easy to verify. Such proof systems are known as *zero-knowledge succinct non-interactive arguments of knowledge* (zk-SNARKs), and are desired when communication is expensive, or the verifier is computationally weak.

Existing zk-SNARK implementations have severe scalability limitations, in terms of space complexity as a function of the size of the computation being proved (e.g., running time of the NP statement's decision program). First, the size of the proving key is quasilinear in the upper bound on the computation size. Second, producing a proof requires "writing down" all intermediate values of the entire computation, and then conducting global operations such as FFTs.

The bootstrapping technique of Bitansky et al. (STOC '13), following Valiant (TCC '08), offers an approach to scalability, by recursively composing proofs: proving statements about acceptance of the proof system's own verifier (and correctness of the program's latest step). Alas, recursive composition of known zk-SNARKs has never been realized in practice, due to enormous computational cost.

Using new elliptic-curve cryptographic techniques, and methods for exploiting the proof systems' field structure and nondeterminism, we achieve the first zk-SNARK implementation that practically achieves recursive proof composition. Our zk-SNARK implementation runs random-access machine programs and produces proofs of their correct execution, on today's hardware, for any program running time. It takes constant time to generate the keys that support *all* computation sizes. Subsequently, the proving process only incurs a constant multiplicative overhead compared to the original computation's time, and an essentially-constant additive overhead in memory. Thus, our zk-SNARK implementation is the first to have a well-defined, albeit low, clock rate of "verified instructions per second".

Keywords: computationally-sound proofs, proof-carrying data, zero-knowledge, elliptic curves.

J.A. Garay and R. Gennaro (Eds.): CRYPTO 2014, Part II, LNCS 8617, pp. 276–294, 2014.

1 Introduction

Non-interactive zero-knowledge proofs of knowledge [BFM88, NY90, BDSMP91] are a powerful tool, studied extensively both in theoretical and applied cryptography. Recently, much research has focused on achieving an additional property, *succinctness*, that requires the proof to be very short and easy to verify. A proof system with this additional property is called a *zero-knowledge Succinct Non-interactive ARgument of Knowledge* (zk-SNARK). Because succinctness is a desirable, sometimes critical, property in numerous security applications, prior work has investigated zk-SNARK implementations. Unfortunately, all implementations to date suffer from severe scalability limitations, due to high space complexity, as we now explain.

1.1 Scalability Limitations of Prior zk-SNARK Implementations

Expensive Preprocessing. 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); the key pair is then published as the proof system's parameters.

Most zk-SNARK constructions [Gro10, Lip12, BCI$^+$13, GGPR13, PGHR13, BCG$^+$13a, Lip13, BCTV14b], including all published implementations [PGHR13, BCG$^+$13a, BCTV14b], require *expensive preprocessing* during key generation. Namely, the key generator takes as input an upper bound on the computation size, e.g., in the form of an explicit NP decision circuit C output by a *circuit generator*; then, the key generator's space complexity, as well as the size of the output proving key, depends at least linearly on this upper bound. Essentially, the circuit C is explicitly laid out and encoded so as to produce the proof system's parameters.

One way to mitigate the costs of expensive preprocessing is to make C universal, i.e., design C so that it can handle more than one choice of program [BCTV14b]. Yet, C *still* depends on upper bounds on the program size and number of execution steps. Moreover, even if key generation is carried out only once per circuit C, the resulting large proving key must be stored, and accessed, *each time a proof is generated*. Prior implementations of zk-SNARKs quickly become space-bound already for modest computation sizes, e.g., with proving keys of over 4 GB for circuits of only 16 million gates [BCTV14b].[1]

Thus, expensive preprocessing severely limits scalability of a zk-SNARK.

Space-Intensive Proof Generation. Related in part to the aforementioned expensive preprocessing, the prover in all published zk-SNARK implementations has large space complexity. Essentially, the proving process requires writing down the *entire* computation (e.g., the evaluation of the circuit C) all at once, and then conduct a global computation (such as Fast Fourier transforms, or multi-exponentiations) based on it. In particular, if C expresses the execution of a program, then proving requires writing down the full trace of intermediate states throughout the program execution.

[1] Even worse, the reported numbers are for "data at rest": the proving key consists of a list of elliptic-curve points, which are *compressed* when not in use. However, when the prover uses the proving key to produce a proof, the points are uncompressed (and represented via projective or Jacobian coordinates), and take about three times as much space in memory.

Tradeoffs are possible, using block-wise versions of the global algorithms, and repeating the computation to reproduce segments of the trace. These decrease the prover's space complexity but significantly increase its time complexity, and thus do not adequately address scalability.

Remark 1. Even when relaxing the goal (by allowing interaction, "theorem batching", or non-zero-knowledge proofs), all published implementations of proof systems for outsourcing NP computations [SBW11, SMBW12, SVP+12, SBV+13, BFR+13] also suffer from both of the above scalability limitations.[2]

1.2 What We Know from Theory

Ideally, we would like to implement a zk-SNARK that does not suffer from either of the scalability limitations mentioned in the previous section, i.e., a zk-SNARK where:

- Key generation is *cheap* (i.e., its running time only depends on the security parameter) and *suffices for all computations* (of polynomial size). Such a zk-SNARK is called **fully succinct**.
- Proof generation is carried out *incrementally*, alongside the original computation, by updating, at each step, a proof of correctness of the computation so far. Such a zk-SNARK is called **incrementally computable**.

Work in cryptography tells us that the above properties can be achieved in theoretical zk-SNARK constructions. Namely, building on the work of Valiant on incrementally-verifiable computation [Val08] and the work of Chiesa and Tromer on proof-carrying data [CT10, CT12], Bitansky et al. [BCCT13] showed how to construct zk-SNARKs that are fully-succinct and incrementally-computable.

Concretely, the approach of [BCCT13] consists of a transformation that takes as input a *preprocessing* zk-SNARK (such as one from existing implementations), and *bootstraps* it, via recursive proof composition, into a new zk-SNARK that is fully-succinct and incrementally-computable. In recursive proof composition, a prover produces a proof about an NP statement that, among other checks, also ensures the accepting computation of the proof system's own verifier. In a zk-SNARK, proof verification is asymptotically cheaper than merely verifying the corresponding NP statement; so recursive proof composition is viable, in theory. In practice, however, this step introduces concretely enormous costs: even if zk-SNARK verifiers can be executed in just a few milliseconds on a modern desktop [PGHR13, BCTV14b], zk-SNARK verifiers still take millions of machine cycles to execute. Hence, known zk-SNARK implementations cannot achieve *even one step* of recursive proof composition in practical time. Thus, whether recursive proof composition can be realized in practice, with any reasonable efficiency, has so far remained an intriguing open question.

Remark 2 (PCPs). Suitably instantiating Micali's "computationally-sound proofs" [Mic00] yields fully-succinct zk-SNARKs. However, it is not known how to also

[2] In contrast, when outsourcing P computations, there are implementations without expensive preprocessing: [CMT12, TRMP12, Tha13] consider low-depth circuits, and [CRR11] consider outsourcing to multiple provers at least one of which is honest.

achieve incremental computation with this approach (without also invoking the aforementioned approach of Bitansky et al. [BCCT13]). Indeed, [Mic00] requires probabilistically-checkable proofs (PCPs) [BFLS91], where one can achieve a prover that runs in quasilinear-time [BCGT13b], but only by requiring space-intensive computations — again due to the need to write down the entire computation and conducting global operations on it.

1.3 Contributions

We present the first prototype implementation that practically achieves recursive composition of zk-SNARKs. This enables us to achieve the following results:

(i) Scalable zk-SNARKs. We present the first implementation of a zk-SNARK that is fully succinct and incrementally computable. Our implementation follows the approach of Bitansky et al. [BCCT13].

Our zk-SNARK works for proving/verifying computations on a general notion of random-access machine. The key generator takes as input a *machine specification*, consisting of settings for random-access memory (number of addresses and number of bits at each address) and a CPU circuit, defining the machine's behavior. The keys sampled by the key generator support proving/verifying computations, of any polynomial length, on this machine. Thus, our zk-SNARK implementation directly supports many architectures (e.g., floating-point processors, SIMD-based processors, etc.) — one only needs to specify memory settings and a CPU circuit.

Compared to the original machine computation, our zk-SNARK only imposes a constant multiplicative overhead in time and an essentially-constant additive overhead in space. Indeed, the proving process steps through the machine's computation, each time producing a new proof that the computation is correct so far, by relying on the prior proof; each proof asserts the satisfiability of a constant-size circuit, and requires few resources in time and space to produce. Our zk-SNARK scales, on today's hardware, to any computation size.

(ii) Proof-Carrying Data. The main tool in [BCCT13]'s approach is *proof-carrying data* (PCD) [CT10, CT12], a cryptographic primitive that encapsulates the security guarantees provided by recursive proof composition. Thus, as a stepping stone towards the aforementioned zk-SNARK implementation, we also achieve the first implementation of PCD, for arithmetic circuits.

(iii) Evaluation on vnTinyRAM**.** We evaluate our zk-SNARK on a specific choice of random-access machine: vnTinyRAM, a simple RISC von Neumann architecture that is supported by the most recent preprocessing zk-SNARK implementation [BCTV14b]. The evaluation confirms our expectations that our approach is slower for small computations but achieves scalability to large computations.

We evaluated our prototype on 16-bit and 32-bit vnTinyRAM with 16 registers (as in [BCTV14b]). For instance, for 32-bit vnTinyRAM, our prototype incrementally proves correct program execution at the cost of 35.5 seconds per program step, using a 64.4 MB proving key and 1,008 MB of additional memory. In contrast, for a T-step program, the system of [BCTV14b] requires roughly $0.05 \cdot T$ seconds, *provided* that roughly $3.1 \cdot T$ MB of main memory is available. Thus for $T > 326$ our system is more

space-efficient, and the savings in space continue to grow as T increases. (These numbers are for an 80-bit security level.)

The Road Ahead. Obtaining scalable zk-SNARKs is but one application of PCD. More generally, PCD enables efficient "distributed theorem proving", which has applications ranging from securing the IT supply chain, to information flow control, and to distributed programming-language semantics [CT10, CT12, CTV13]. Now that a first prototype of PCD has been achieved, these applications are waiting to be explored in practice.

1.4 Summary of Challenges and Techniques

As we recall in Section 2, bootstrapping zk-SNARKs involves two main ingredients: a collision-resistant hash function and a preprocessing zk-SNARK. Practical implementations of both ingredients exist. So one may conclude that "practical bootstrapping" is merely a matter of stitching together implementations of these two ingredients. As we now explain, this conclusion is mistaken, because bootstrapping a zk-SNARK in practice poses several challenges that must be tackled in order to obtain any reasonable efficiency.

Common Theme: Leverage Field Structure. The techniques that we employ to overcome efficiency barriers leverage the fact that the "native" NP language whose membership is proved/verified by the zk-SNARK is the satisfiability of $\mathbb{F}$-arithmetic circuits, for a certain finite field $\mathbb{F}$. While any NP statement can be reduced to $\mathbb{F}$-arithmetic circuits, the proof system is most efficient for statements expressible as $\mathbb{F}$-arithmetic circuits of small size. Prior work only partially leveraged this fact, by using circuits that conduct large-integer arithmetic or "pack" bits into field elements for non-bitwise checks (e.g., equality) [PGHR13, BCG$^+$13a, BFR$^+$13, BCTV14b]. In this paper, we go further and, for improved efficiency, use circuits that conduct *field operations*.

Challenge: How to Efficiently "Close the Loop"? By far the most prominent challenge is efficiently "closing the loop". In the bootstrapping approach, each step requires proving a statement that (i) verifies the validity of previous zk-SNARK proofs; and (ii) checks another execution step. For recursive composition, this statement needs to be expressed as an $\mathbb{F}$-arithmetic circuit C_{pcd}, so that it can be proved using the very same zk-SNARK. In particular, we need to *implement the verifier V as an $\mathbb{F}$-arithmetic circuit C_V* (a subcircuit of C_{pcd}).

In principle, constructing C_V is possible, because circuits are a universal model of computation. In fact, not just in principle: much research has been devoted to improve the efficiency and functionality of circuit generators in practice [SVP$^+$12, BCGT13a, SBV$^+$13, PGHR13, BCG$^+$13a, BCTV14b]. Hence, a reasonable approach to construct C_V is to apply a suitable circuit generator to a suitable software implementation of V.

However, such an approach is likely to be inefficient. Circuit generators strive to support complex program computations, by providing ways to efficiently handle data-dependent control flow, memory accesses, and so on. Instead, verifiers in preprocessing zk-SNARK constructions are "circuit-like" programs, consisting of few pairing-based arithmetic checks that do not use complex data-dependent control flow or memory accesses.

Thus, we want to avoid circuit generators, and somehow directly construct C_V so that its size is not huge. As we shall explain (see Section 3), this is not merely a programmatic difficulty, but there are *mathematical obstructions* to constructing C_V efficiently.

Main Technique: PCD-Friendly Cycles of Elliptic Curves. In our underlying preprocessing zk-SNARK, the verifier V consists mainly of operations in an elliptic curve over a field $\mathbb{F}'$, and is thus expressed, most efficiently, as a $\mathbb{F}'$-arithmetic circuit. We observe that if this field $\mathbb{F}'$ is the same as the aforementioned native field $\mathbb{F}$ of the zk-SNARK's statement, then recursive composition can be orders of magnitude more efficient than otherwise. Unfortunately, as we shall explain, the "field matching" $\mathbb{F} = \mathbb{F}'$ is mathematically impossible.

In contrast, we show how to circumvent this obstruction by using multiple, suitably-chosen elliptic curves, that lie on a *PCD-friendly cycle*. For example, a PCD-friendly 2-cycle consists of two curves such that the (prime) size of the base field of one curve equals the group order of the other curve, and vice versa. Our implementation uses a PCD-friendly cycle of elliptic curves (found at a great computational expense) to attain zk-SNARKs that are *tailored* for recursive proof composition.

Additional Technique: Nondeterministic Verification of Pairings. The zk-SNARK verifier involves, more specifically, several pairing-based checks over its elliptic curve. Yet, each pairing evaluation is very expensive, if not carefully performed. To further improve efficiency, we exploit the fact that the zk-SNARK supports NP statements, and provide a hand-optimized circuit implementation of the zk-SNARK verifier that leverages nondeterminism for improved efficiency. For instance, in our construction, we make heavy use of *affine* coordinates for both curve arithmetic and divisor evaluations [LMN10], because these are particularly efficient to *verify* (as opposed to *computing*, for which projective or Jacobian coordinates are known to be faster).

Challenge: How to Efficiently Verify Collision-Resistant Hashing? Bootstrapping zk-SNARKs uses, at multiple places, a collision-resistant hash function H and an arithmetic circuit C_H for verifying computations of H. If not performed efficiently, this would be another bottleneck.

For instance, the aforementioned circuit C_{pcd}, besides verifying prior zk-SNARK proofs, is also tasked with verifying one step of machine execution. This involves not only checking the CPU execution but also the validity of loads and stores to random-access memory, done via memory-checking techniques based on Merkle trees [BEG$^+$91, BCGT13a]. Thus C_{pcd} also needs to have a subcircuit to check Merkle-tree authentication paths. Constructing such circuits is straightforward, given a circuit C_H for verifying computations of H. But the main question here is how to pick H so that C_H can be small. Indeed, if random-access memory consists of A addresses, then checking an authentication path requires at least $\lceil \log A \rceil \cdot |C_H|$ gates. If C_H is large, this subcircuit *dwarfs* the CPU, and "wastes" most of the size of C_{pcd} for a single load/store.

Merely picking some standard choice of hash function H (e.g., SHA-256 or Keccak) yields C_H with tens of thousands of gates [PGHR13, BCG$^+$14], making hash verifications very expensive. Is this inherent?

Additional Technique: Field-Specific Hashes. We select a hash H that is tailored to efficient verification in the field $\mathbb{F}$. In our setting, $\mathbb{F}$ has prime order p, so its additive

group is isomorphic to $\mathbb{Z}_p$. Thus, a natural approach is to let H be a *modular subset-sum* function over $\mathbb{Z}_p$. For suitable parameter choices and for random coefficients, subset-sum functions are collision-resistant [Ajt96, GGH96]. In this paper we base all of our collision-resistant hashing on suitable subset sums, and thereby greatly reduce the burden of hashing.[3]

1.5 Roadmap

The rest of this paper is organized as follows. In Section 2 we recall the main ideas of [BCCT13]'s approach. In Section 3, we discuss our construction of preprocessing zk-SNARKs that are tailored for efficient recursive composition of proofs; due to space constraints, we leave the other discussions (construction of proof-carrying data and scalable zk-SNARK) to the full version of this paper [BCTV14a]. In Section 4, we evaluate our system on the random-access machine vnTinyRAM.

2 Preliminaries

2.1 Preprocessing zk-SNARKs for Arithmetic Circuits

Given a field $\mathbb{F}$, the *circuit satisfaction problem* of an $\mathbb{F}$-arithmetic circuit $C\colon \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^l$ is defined by the relation $\mathcal{R}_C = \{(x, a) \in \mathbb{F}^n \times \mathbb{F}^h : C(x, a) = 0^l\}$; its language is $\mathcal{L}_C = \{x \in \mathbb{F}^n : \exists\, a \in \mathbb{F}^h,\ C(x, a) = 0^l\}$.

A **preprocessing zk-SNARK** for $\mathbb{F}$-arithmetic circuit satisfiability (see, e.g., [BCI$^+$13]) 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 $\mathcal{L}_C$, and anyone can use the vk to check these proofs. Namely, given pk and any $(x, a) \in \mathcal{R}_C$, the honest prover $P(\mathsf{pk}, x, a)$ produces a proof π attesting that $x \in \mathcal{L}_C$; the verifier $V(\mathsf{vk}, x, \pi)$ checks that π is a valid proof for $x \in \mathcal{L}_C$. A proof π is a proof of knowledge, as well as a (statistical) zero-knowledge proof. The succinctness property requires that π has length $O_\lambda(1)$ and V runs in time $O_\lambda(|x|)$, where O_λ hides a (fixed) polynomial in λ.

See the full version of this paper for details [BCTV14a].

2.2 Proof-Carrying Data

Proof-carrying data (PCD) [CT10, CT12] is a cryptographic primitive that encapsulates the security guarantees obtainable via recursive composition of proofs. Since recursive proof composition naturally involves multiple (physical or virtual) parties, PCD is phrased in the language of a dynamically-evolving *distributed computation* among mutually-untrusting computing nodes, who perform local

[3] We note that subset-sum functions were also used in [BFR$^+$13], but, crucially, they were *not* tailored to the field. This is a key difference in usage and efficiency. (E.g., our hash function can be verified in ≤ 300 gates, while [BFR$^+$13] report 13,000.)

computations, based on local data and previous messages, and then produce output messages. Given a *compliance predicate* Π to express local checks, the goal of PCD is to ensure that any given message z in the distributed computation is Π*-compliant*, i.e., is consistent with a history in which each node's local computation satisfies Π. This formulation includes as special cases incrementally-verifiable computation [Val08] and targeted malleability [BSW12].

Concretely, a proof-carrying data (PCD) system is a triple of polynomial-time algorithms $(\mathbb{G}, \mathbb{P}, \mathbb{V})$, called *key generator*, *prover*, and *verifier*. The key generator $\mathbb{G}$ is given as input a predicate Π (specified as an arithmetic circuit), and outputs a proving key pk and a verification key vk; these keys allow anyone to prove/verify that a piece of data z is Π-compliant. This is achieved by attaching a short and easy-to-verify proof to each piece of data. Namely, given pk, received messages $\boldsymbol{z}_{\mathsf{in}}$ with proofs $\boldsymbol{\pi}_{\mathsf{in}}$, local data z_{loc}, and a claimed outgoing message z, $\mathbb{P}$ computes a new proof π to attach to z, which attests that z is Π-compliant; the verifier $\mathbb{V}(\mathsf{vk}, z, \pi)$ verifies that z is Π-compliant. A proof π is a proof of knowledge, as well as a (statistical) zero-knowledge proof; succinctness requires that π has length $O_\lambda(1)$ and $\mathbb{V}$ runs in time $O_\lambda(|z|)$.

Finally, note that since Π is expressed as an $\mathbb{F}$-arithmetic circuit for a given field $\mathbb{F}$, the size of messages and local data are fixed; we denote these sizes by $n_{\mathsf{msg}}, n_{\mathsf{loc}} \in \mathbb{N}$. Similarly, the number of input messages is also fixed; we call this the *arity*, and denote it by $s \in \mathbb{N}$. Moreover, for convenience, Π also takes as input a flag $b_{\mathsf{base}} \in \{0,1\}$ denoting whether the node has no predecessors (i.e., b_{base} is a "base-case" flag). Overall, Π takes an input $(z, z_{\mathsf{loc}}, \boldsymbol{z}_{\mathsf{in}}, b_{\mathsf{base}}) \in \mathbb{F}^{n_{\mathsf{msg}}} \times \mathbb{F}^{n_{\mathsf{loc}}} \times \mathbb{F}^{s \cdot n_{\mathsf{msg}}} \times \mathbb{F}$.

See the full version of this paper for details [BCTV14a].

2.3 The Bootstrapping Approach

Our implementation follows [BCCT13], which we now review. The approach consists of a transformation that, on input a preprocessing zk-SNARK and a collision-resistant hash function, outputs a scalable zk-SNARK. Thus, the input zk-SNARK is *bootstrapped* into one with improved scalability properties.

So fix a preprocessing zk-SNARK (G, P, V) and collision-resistant function H. The goal is to construct a fully-succinct incrementally-computable zk-SNARK $(G^\star, P^\star, V^\star)$ for proving/verifying the correct execution on a given random-access machine $\mathbf{M}$. Informally, we describe the transformation in four steps.

Step 1: from zk-SNARKs to PCD. The first step, independent of $\mathbf{M}$, is to construct a PCD system $(\mathbb{G}, \mathbb{P}, \mathbb{V})$, by using the zk-SNARK (G, P, V). This step involves recursive composition of zk-SNARK proofs.

Step 2: Delegate the Machine's Memory. The second step is to reduce the footprint of the machine $\mathbf{M}$, by delegating its random-access memory to an untrusted storage, via standard memory-checking techniques based on Merkle trees [BEG+91, BCGT13a]. We thus modify $\mathbf{M}$ so that its "CPU" receives values loaded from memory as nondeterministic guesses, along with corresponding authentication paths that are checked against the root of a Merkle tree based on the hash function H. Thus, $\mathbf{M}$'s state only

consists of a (short) CPU state, and a (short) root of the Merkle tree that "summarizes" memory.[4]

Step 3: Design a Predicate $\Pi_{\mathbf{M},H}$ for Step-Wise Verification. The third step is to design a compliance predicate $\Pi_{\mathbf{M},H}$ that ensures that the only $\Pi_{\mathbf{M},H}$-compliant messages z are the ones that result from the correct execution of the (modified) machine $\mathbf{M}$, one step at a time; this is analogous to the notion of incremental computation [Val08]. Crucially, because $\Pi_{\mathbf{M},H}$ is only asked to verify one step of execution at a time, we can implement $\Pi_{\mathbf{M},H}$'s requisite checks with a circuit of merely constant size.

Step 4: Construct New Proof System. The new zk-SNARK $(G^\star, P^\star, V^\star)$ is constructed as follows. The new key generator $G^\star$ is set to the PCD generator $\mathbb{G}$ invoked on $\Pi_{\mathbf{M},H}$. The new prover $P^\star$ uses the PCD prover $\mathbb{P}$ to prove correct execution of $\mathbf{M}$, one step at a time and conducting the incremental distributed computation "in his head". The new verifier $V^\star$ simply uses the PCD verifier $\mathbb{V}$ to verify $\Pi_{\mathbf{M},H}$-compliance. In sum, since $\Pi_{\mathbf{M},H}$ is small and suffices for all computations, the new zk-SNARK is scalable: it is fully succinct; moreover, because the new prover computes a proof for each new step based on the previous one, it is also incrementally computable. (See the full version of this paper for definitions of these properties [BCTV14a].)

Our goal is to realize the above approach in a practical implementation.

Security of Recursive Proof Composition. Security in [BCCT13] is proved by using the *proof-of-knowledge property* of zk-SNARKs; we refer the interested reader to [BCCT13] for details. One aspect that must be addressed from a theoretical standpoint is the *depth* of composition. Depending on assumption strength, one may have to recursively compose proofs in "proof trees above the message chain", rather than along the chain. From a practical perspective we make the heuristic assumption that depth of composition does not affect security of the zk-SNARK, because no evidence suggests otherwise for the constructions that we use.

3 PCD-Friendly Preprocessing zk-SNARKs

We first construct preprocessing zk-SNARKs that are tailored for efficient recursive composition of proofs.

3.1 PCD-Friendly Cycles of Elliptic Curves

Let $\mathbb{F}$ be a finite field, and (G, P, V) a preprocessing zk-SNARK for $\mathbb{F}$-arithmetic satisfiability. The idea of recursive proof composition is to prove/verify satisfiability of an $\mathbb{F}$-arithmetic circuit C_{pcd} that checks the validity of previous proofs (among other things). Thus, we need to implement the verifier V as an $\mathbb{F}$-arithmetic circuit C_V, to be used as a sub-circuit of C_{pcd}.

[4] Similarly to [BCCT13] and our realization thereof, Braun et al. [BFR+13] leverage memory-checking techniques based on Merkle trees [BEG+91] for enabling a circuit to "securely" load from and store to an untrusted storage. However, the systems' goals (batched verification of MapReduce computations in a 2-move protocol) and techniques are different (cf. Footnote 3).

How to write C_V depends on the algorithm of V, which in turn depends on which elliptic curve is used to instantiate the pairing-based zk-SNARK. For prime r, in order to prove statements about $\mathbb{F}_r$-arithmetic circuit satisfiability, one instantiates (G, P, V) using an elliptic curve E defined over some finite field $\mathbb{F}_q$, where the group $E(\mathbb{F}_q)$ of $\mathbb{F}_q$-rational points has order $r = \#E(\mathbb{F}_q)$ (or, more generally, r divides $\#E(\mathbb{F}_q)$). Then, all of V's arithmetic computations are over $\mathbb{F}_q$, or extensions of $\mathbb{F}_q$ up to degree k, where k is the embedding degree of E with respect to r (i.e., the smallest integer k such that r divides $q^k - 1$). We motivate our approach by first describing two "failed attempts".

Attempt #1: Pick Curve with $q = r$. Ideally, we would like to select a curve E with $q = r$, so that V's arithmetic is over the *same field* for which V's native NP language is defined. Unfortunately, this cannot happen: the condition that E has embedding degree k with respect to r implies that r divides $q^k - 1$, which implies that $q \neq r$. The same implication holds even if $E(\mathbb{F}_q)$ has a non-prime order n and the prime r (with respect to which k is defined) only divides n. So, while appealing, this idea cannot even be instantiated.

Attempt #2: Long Arithmetic. Since we are stuck with $q \neq r$, we may consider doing "long arithmetic": simulating $\mathbb{F}_q$ operations via $\mathbb{F}_r$ operations, by working with bit chunks to perform integer arithmetic, and modding out by q when needed. Alas, having to work at the "bit level" implies a blowup on the order of $\log q$ compared to native arithmetic. So, while this approach can at least be instantiated, it is very expensive.

Our Approach: Cycle through Multiple Curves. We formulate, and instantiate, a new property for elliptic curves that enables us to completely circumvent long arithmetic, even with $q \neq r$. In short, our idea is to base recursive proof composition, not on a single zk-SNARK, but on *multiple* zk-SNARKs, each instantiated on a different elliptic curve, that *jointly* satisfy a special property.

For the simplest case, suppose we have two primes q_α and q_β, and elliptic curves $E_\alpha/\mathbb{F}_{q_\alpha}$ and $E_\beta/\mathbb{F}_{q_\beta}$ such that $q_\alpha = \#E_\beta(\mathbb{F}_{q_\beta})$ and $q_\beta = \#E_\alpha(\mathbb{F}_{q_\alpha})$, i.e., the size of the base field of one curve equals the group order of the other curve, and vice versa. We then construct two preprocessing zk-SNARKs $(G_\alpha, P_\alpha, V_\alpha)$ and $(G_\beta, P_\beta, V_\beta)$, respectively instantiated on the two curves $E_\alpha/\mathbb{F}_{q_\alpha}$ and $E_\beta/\mathbb{F}_{q_\beta}$.

Now note that $(G_\alpha, P_\alpha, V_\alpha)$ works for $\mathbb{F}_{q_\beta}$-arithmetic circuit satisfiability, but all of V_α's arithmetic computations are over $\mathbb{F}_{q_\alpha}$ (or extensions thereof); while $(G_\beta, P_\beta, V_\beta)$ works for $\mathbb{F}_{q_\alpha}$-arithmetic circuits, but V_β's arithmetic computations are over $\mathbb{F}_{q_\beta}$ (or extensions thereof). Instead of having each zk-SNARK handle statements about its *own* verifier, as in the prior attempts (i.e., writing V_α as a $\mathbb{F}_{q_\beta}$-arithmetic circuit, or V_β as a $\mathbb{F}_{q_\alpha}$-arithmetic circuit), we instead let each zk-SNARK handle statements about the verifier of the *other* zk-SNARK. That is, we write V_α as a $\mathbb{F}_{q_\alpha}$-arithmetic circuit C_{V_α}, and V_β as a $\mathbb{F}_{q_\beta}$-arithmetic circuit C_{V_β}.

We can then perform recursive proof composition by *alternating* between the two proof systems. Roughly, one can use P_α to prove successful verification of a proof by C_{V_β} and, conversely, P_β to prove successful verification of a proof by C_{V_α}. Doing so in alternation ensures that fields "match up", and no long arithmetic is needed. (This sketch omits key technical details; see the full version of this paper [BCTV14a].)

Since E_α and E_β facilitate constructing PCD, we say that (E_α, E_β) is a *PCD-friendly 2-cycle of elliptic curves*. More generally, the idea extends to cycling through ℓ curves satisfying this definition:

Definition 1. *Let $E_0, \ldots, E_{\ell-1}$ be elliptic curves, respectively defined over finite fields $\mathbb{F}_{q_0}, \ldots, \mathbb{F}_{q_{\ell-1}}$, with each q_i a prime. We say that $(E_0, \ldots, E_{\ell-1})$ is a* **PCD-friendly cycle** *of length ℓ if each E_i is pairing friendly and, moreover, $\forall\, i \in \{0, \ldots, \ell-1\},\ q_i = \#E_{i+1 \bmod \ell}(\mathbb{F}_{q_{i+1 \bmod \ell}})$.*

To our knowledge this notion has not been explicitly sought before. Though, fortunately, a family that satisfies this notion is already known, as discussed in the next subsection.

3.2 Two-Cycles Based on MNT Curves

We construct pairs of elliptic curves, E_4 and E_6, that form PCD-friendly 2-cycles (E_4, E_6). These are MNT curves [MNT01] of embedding degrees 4 and 6. Our construction also ensures that E_4 and E_6 are sufficiently *2-adic* (see below), a desirable property for efficient implementations of preprocessing zk-SNARKs.

MNT Curves and the KT Correspondence. Miyaji, Nakabayashi, and Takano [MNT01] characterized prime-order elliptic curves with embedding degrees $k = 3{,}4{,}6$; such curves are now known as *MNT curves*. Given an elliptic curve E defined over a prime field $\mathbb{F}_q$, they gave necessary and sufficient conditions on the pair (q, t), where t is the *trace* of E over $\mathbb{F}_q$, for E to have embedding degree $k = 3{,}4{,}6$. We refer to an MNT curve with embedding degree k as an MNTk curve. Karabina and Teske [KT08] proved an explicit 1-to-1 correspondence between MNT4 and MNT6 curves:

Theorem 1 ([KT08]). *Let $n, q > 64$ be primes. Then the following two conditions are equivalent: (i) n and q represent an elliptic curve $E_4/\mathbb{F}_q$ with embedding degree $k = 4$ and $n = \#E(\mathbb{F}_q)$; (ii) n and q represent an elliptic curve $E_6/\mathbb{F}_n$ with embedding degree $k = 6$ and $q = \#E(\mathbb{F}_n)$.*

PCD-Friendly 2-Cycles on MNT Curves. The above theorem implies that:

Each MNT6 curve lies on a PCD-friendly 2-cycle with the corresponding MNT4 curve (and vice versa).

Thus, a PCD-friendly 2-cycle can be obtained by constructing an MNT4 curve and its corresponding MNT6 curve. Next, we explain at high level how this can be done.

Constructing PCD-Friendly 2-Cycles. First, we recall the only known method to construct MNTk curves [MNT01]. It consists of two steps:

- Step I: *curve discovery*. Find suitable $(q, t) \in \mathbb{N}^2$ such that there exists an ordinary elliptic curve $E/\mathbb{F}_q$ of prime order $n := q + 1 - t$ and embedding degree k.
- Step II: *curve construction*. Starting from (q, t), use the *Complex-Multiplication method* (CM method) [AM93] to compute the equation of E over $\mathbb{F}_q$.

The complexity of Step II depends on the *discriminant* D of E, which is the square-free part of $4q - t^2$. At present, the CM method is feasible for discriminants D up to size 10^{16} [Sut12]. Thus, Step I is conducted in a way that results in candidate parameters

(q,t) inducing relatively-small discriminants, to aid Step II. (Instead, "most" (q,t) induce a discriminant D of size $\sqrt{q}$, which is too large to handle.) Concretely, [MNT01] derived, for $k \in \{3,4,6\}$ and discriminant D, Pell-type equations whose solutions yield candidate parameters (q,t) for MNTk curves $E/\mathbb{F}_q$ of trace t and discriminant D. So Step I can be performed by iteratively solving the *MNTk Pell-type equation*, for increasing discriminant size, until a suitable (q,t) is found.

The above strategy can be extended, in a straightforward way, to construct PCD-friendly 2-cycles. First perform Step I to obtain suitable parameters (q_4,t_4) for an MNT4 curve $E_4/\mathbb{F}_{q_4}$; the parameters (q_6,t_6) for the corresponding MNT6 curve $E_6/\mathbb{F}_{q_6}$ are $q_6 := q_4 + 1 - t_4$ and $t_6 := 2 - t_4$. Then perform Step II for (q_4,t_4) to compute the equation of E_4, and then also for (q_6,t_6) to compute that of E_6. The complexity in both cases is the same: one can verify that E_4 and E_6 have the same discriminant. The two curves E_4 and E_6 form a PCD-friendly 2-cycle (E_4,E_6).

Suitable Cycle Parameters. We now explain what "suitable (q_4,t_4)" means in our context, by specifying a list of additional properties that we wish a PCD-friendly cycle to satisfy.

- *Bit lengths.* In a 2-cycle (E_4,E_6), the curve E_4 is "less secure" than E_6, because E_4 has embedding degree 4 while E_6 has embedding degree 6. Thus, we use E_4 to set lower bounds on bit lengths. Since we aim at a security level of 80 bits, we need $r_4 \geq 2^{160}$ and $q_4 \geq 2^{240}$ (so that $\sqrt{r_4} \geq 2^{80}$ and $q_4^4 \geq 2^{960}$ [FST10]). Since $\log r_4 \approx \log q_4$ for MNT4 curves, we only need to ensure that q_4 has 240 bits.[5]
- *Towering friendliness.* We restrict our focus to moduli q_4 and q_6 that are *towering friendly* (i.e., congruent to 1 modulo 6) [BS10]; this improves the efficiency of arithmetic in $\mathbb{F}_{q_4}^4$ and $\mathbb{F}_{q_6}^6$ (and their subfields).
- *2-adicity.* As discussed in [BCG+13a, BCTV14b], if a pairing-based preprocessing zk-SNARK (G,P,V) is instantiated with an elliptic curve $E/\mathbb{F}_q$ of prime order r (or with $\#E(\mathbb{F}_q)$ divisible by a prime r), it is important, for efficiency reasons, that $r-1$ is divisible by a large power of 2, i.e., $\nu_2(r-1)$ is large. (Recall that $\nu_2(n)$, the 2-adic order of n, is the largest power of 2 dividing n.) Concretely, if G is invoked on an $\mathbb{F}_r$-arithmetic circuit C, it is important that $\nu_2(r-1) \geq \lceil \log |C| \rceil$. We call $\nu_2(r-1)$ the *2-adic order of E*, or the *2-adicity of E*.
  So let ℓ_4 and ℓ_6 be the target values for $\nu_2(r_4-1)$ and $\nu_2(r_6-1)$. One can verify that, for any MNT-based PCD-friendly 2-cycle (E_4,E_6), it holds that $\nu_2(r_4-1) = 2 \cdot \nu_2(r_6-1)$; in other words, E_4 is always "twice as 2-adic" as E_6. Thus, to achieve the target 2-adic orders, it suffices to ensure that $\nu_2(r_4-1) \geq \max\{\ell_4, 2\ell_6\}$ (where, as before, $r_4 := q_4 + 1 - t_4$). As we shall see it will suffice to take $\nu_2(r_4-1) \geq 34$.

Of the above properties, the most restrictive one is 2-adicity, because it requires seeing enough curves until, "by sheer statistics", one finds (q_4,t_4) with a high-enough value for $\nu_2(r_4-1)$. Collecting enough samples is costly because, as discriminant size increases, the density of MNT curves decreases: empirically, one finds that the number MNT curves with discriminant $D \leq N$ is (approximately) less than $\sqrt{N}$ [KT08].

[5] Alas, since E_4 has a low embedding degree, the ECDLP in $E(\mathbb{F}_{q_4})$ and DLP in $\mathbb{F}_{q_4}^4$ are "unbalanced": the former provides 120 bits of security, while the latter only 80. Moreover, the same is true for E_6: the ECDLP in $E(\mathbb{F}_{q_6})$ provides 120 bits of security, while the DLP in $\mathbb{F}_{q_4}^6$ only 80. Finding PCD-friendly cycles without these inefficiencies is an open problem.

An Extensive Computation for a Suitable Cycle. Overall, finding and constructing a suitable cycle required a substantial computational effort.

- *Cycle discovery.* In order to find suitable parameters for a cycle, we explored a large space: all discriminants up to $1.1 \cdot 10^{15}$, requiring about 610,000 core-hours on a large cluster of modern x86 servers. Our search algorithm is a modification of [KT08, Algorithm 3]. Among all the 2-cycles that we found, we selected parameters (q_4, t_4) and (q_6, t_6) for a 2-cycle (E_4, E_6) of curves such that: (i) q_4, q_6 each have 298 bits; (ii) q_4, q_6 are towering friendly; and (iii) $\nu_2(r_4 - 1) = 34$ and $\nu_2(r_6 - 1) = 17$. The bit length of q_4, q_6 is higher than the lower bound of 240; we entail this cost so to pick a rare cycle with high 2-adicity, which helps the zk-SNARK's efficiency more than the slowdown incurred by the higher bit length.
- *Cycle construction.* Both E_4 and E_6 have discriminant 614144978799019, whose size requires state-of-the-art techniques in the CM method [Sut11, ES10, Sut12] in order to explicitly construct the curves.[6]

Below, we report the parameters and equations for the 2-cycle (E_4, E_6) that we selected.

$E_4/\mathbb{F}_{q_4} : y^2 = x^3 + A_4 x + B_4$ where

$A_4 = 2,$

$B_4 = 42389453652668417828941601153388824002931810367389600280334154412405474501934079536084168 5,$

$q_4 = 47592228616926132575334924965304845154512487924269472539555512857621026281795580048375808 1.$

$E_6/\mathbb{F}_{q_6} : y^2 = x^3 + A_6 x + B_6$ where

$A_6 = 11,$

$B_6 = 10670008051085173567796731963258535225645425120136758789018598936293600026260666846952307 4,$

$q_6 = 47592228616926132575334924965304845154512487855282351555326773573916464730740849055996313 7.$

Security. One may wonder if curves lying on PCD-friendly cycles are weak (e.g., in terms of DL hardness). Yet, MNT4 and MNT6 curves of suitable parameters are widely believed to be secure, and they *all* fall in PCD-friendly 2-cycles. The additional requirement of high 2-adicity is not known to cause weakness either.

3.3 A Matched Pair of Preprocessing zk-SNARKs

Based on the cycle (E_4, E_6), we designed and constructed two preprocessing zk-SNARKs for arithmetic circuit satisfiability: (G_4, P_4, V_4) based on the curve E_4, and (G_6, P_6, V_6) on E_6. The software implementation follows [BCTV14b], the fastest preprocessing zk-SNARK implementation for circuits at the time of writing. We thus adapt the techniques in [BCTV14b] to our algebraic setting, which consists of the two MNT curves E_4 and E_6, and achieve efficient implementations of (G_4, P_4, V_4) and (G_6, P_6, V_6).

The implementation itself entails many algorithmic and engineering details, and we refer the reader to [BCTV14b] for a discussion of these techniques. We only provide a

[6] The authors are grateful to Andrew V. Sutherland for generous help in running the CM method on such a large discriminant.

high-level efficiency comparison between the preprocessing zk-SNARK of [BCTV14b] based on Edwards curves (also at 80-bit security), and our implementations of (G_4, P_4, V_4) and (G_6, P_6, V_6); see the full version of this paper. Our implementation is slower, because of two main reasons: (i) MNT curves do not enjoy advantageous properties that Edwards curves do; and (ii) the modulus sizes are larger (298 bits in our case vs. 180 bits in [BCTV14b]). On the other hand, the fact that MNT curves lie on a PCD-friendly 2-cycle is crucial for the PCD construction described next.

4 Evaluation on vnTinyRAM

We evaluate our scalable zk-SNARK when the given random-access machine $\mathbf{M}$ equals vnTinyRAM, a simple RISC von Neumann architecture [BCTV14b, BCG+13b]. For comparison, we also compare [BCTV14b]'s preprocessing zk-SNARK (which also supports vnTinyRAM) with our scalable zk-SNARK.

We ran our experiments on a desktop PC with a 3.40 GHz Intel Core i7-4770 CPU and 16 GB of RAM available. Unless otherwise specified, all times are in single-thread mode; as for our multi-core experiments, we enabled one thread for each of the CPU's 4 cores (for a total of 4 threads).

Recalling vnTinyRAM. The architecture vnTinyRAM is parametrized by the *word size*, denoted w, and the *number of registers*, denoted k. In terms of instructions, vnTinyRAM includes load and store instructions for accessing random-access memory (in byte or word blocks), as well as simple integer, shift, logical, compare, move, and jump instructions. Thus, vnTinyRAM can efficiently implement control flow, loops, subroutines, recursion, and so on. Complex instructions (e.g., floating-point arithmetic) are not directly supported and can be implemented "in software". See the full version of this paper for how vnTinyRAM can be expressed in our random-access machine formalism (i.e., given w, k, how to construct $\mathbf{M}$ to express w-bit vnTinyRAM with k registers).

Costs on vnTinyRAM. The performance of our zk-SNARK $(G^\star, P^\star, V^\star)$ on vnTinyRAM is easy to characterize, because it is determined by few quantities. For the key generator $G^\star$, the relevant quantities are:

- the constant time and space complexity of $G^\star$, when given as input a description of vnTinyRAM; and
- the constant sizes of the generated proving key pk and verification key vk.

For the proving algorithm $P^\star$, which proceeds step by step alongside the original computation, they are:

- the constant time necessary to incrementally compute the new (constant-size) proof at each step; and
- the constant space needed to compute the new proof (on top of the space needed by the original program).[7]

[7] The prover also needs to store the Merkle tree's intermediate hashes, which incurs a linear overhead in the program's space complexity. Since this overhead is small, and can even be reduced by saving only the high levels of the Merkle tree (and recomputing, "on demand", the local neighborhood of accessed leaves), we focus on the additive overhead needed for proving.

Finally, the verifier $V^\star$ takes as input a program $\mathcal{P}$ and a time bound T, and runs in time $O(|\mathcal{P}| + \log T)$; in our implementation, we fix $T \leq 2^{300}$ (plenty enough), so that $V^\star$ runs in time $O(|\mathcal{P}|)$.

In Figure 1, we report our measurements for two settings of vnTinyRAM: $(w, k) = (16{,}16)$ and $(w, k) = (32{,}16)$, i.e., 16-bit and 32-bit vnTinyRAM with 16 registers. (The same settings as in [BCTV14b].)

	16-bit vnTinyRAM $(w,k) = (16{,}16)$		32-bit vnTinyRAM $(w,k) = (32{,}16)$	
key generator $G^\star$				
	TIME			
	1 thread	4 threads	1 thread	4 threads
total	44.5 s	15.9 s	53.8 s	19.4 s
	SPACE			
memory	1.0 GB	1.2 GB	1.1 GB	1.4 GB
pk size	51.5 MB		64.4 MB	
vk size	1.3 kB		1.3 kB	
prover $P^\star$				
	TIME			
	1 thread	4 threads	1 thread	4 threads
per step	33.1 s	11.5 s	35.5 s	12.1 s
	SPACE			
memory	0.9 GB	1.2 GB	1.1 GB	1.3 GB
proof	374 B		374 B	
verifier $V^\star$				
	TIME			
$\lvert\mathcal{P}\rvert = 10$	23.6 ms		24.3 ms	
$\lvert\mathcal{P}\rvert = 10^2$	24.1 ms		24.9 ms	
$\lvert\mathcal{P}\rvert = 10^3$	30.1 ms		31.1 ms	
$\lvert\mathcal{P}\rvert = 10^4$	91.0 ms		94.1 ms	
in general	$\approx (23.48 + 0.00674\lvert\mathcal{P}\rvert)$ ms		$\approx (24.17 + 0.00698\lvert\mathcal{P}\rvert)$ ms	

Fig. 1. Performance of our scalable zk-SNARK on 16-bit and 32-bit vnTinyRAM

Comparison with [BCTV14b]. In Figure 2, we compare the efficiency of [BCTV14b]'s preprocessing zk-SNARK and our scalable zk-SNARK, for a (random) program $\mathcal{P}$ of 10^4 instructions, as a function of T (the number of vnTinyRAM computation steps).

		key generator		key sizes		prover		verifier
		TIME	SPACE	|pk|	|vk|	TIME	SPACE	TIME
16-bit vnTinyRAM	[BCTV14b]	$0.09 \cdot T$ s	$1.8 \cdot T$ MB	$0.3 \cdot T$ MB	40.4 kB	$0.04 \cdot T$ s	$1.9 \cdot T$ MB	24.2 ms
$(w,k) = (16{,}16)$	**this work**	44.5 s	933 MB	51.5 MB	1.3 kB	$33.1 \cdot T$ s	873 MB	91.0 ms
32-bit vnTinyRAM	[BCTV14b]	$0.14 \cdot T$ s	$3.0 \cdot T$ MB	$0.4 \cdot T$ MB	80.3 kB	$0.05 \cdot T$ s	$3.1 \cdot T$ MB	41.0 ms
$(w,k) = (32{,}16)$	**this work**	53.8 s	1,082 MB	64.4 MB	1.3 kB	$35.5 \cdot T$ s	1,008 MB	94.1 ms

Fig. 2. Comparison between [BCTV14b]'s preprocessing zk-SNARK and our scalable zk-SNARK

The (approximate) asymptotic efficiency for [BCTV14b] was obtained by linearly interpolating [BCTV14b]'s measurements (which were collected on a machine with similar characteristics as our benchmarking machine). As for our measurements, we use the relevant numbers from Figure 1.

Conclusion. Our experiments demonstrate that, as expected, our approach is slower for small computations but, on the other hand, offers scalability to large computations by avoiding any space-intensive computations.

Indeed, [BCTV14b] (as well as other preprocessing zk-SNARK implementations [PGHR13, BCG+13a]) require space-intensive computations to maintain their efficiency. As T grows, such approaches simply run out of memory, and must resort to "computing in blocks", sacrificing time complexity.

In contrast, our zk-SNARK, while requiring more time per execution step, merely requires a constant amount of memory to prove any number of execution steps. In particular, our zk-SNARK becomes more space-efficient than [BCTV14b]'s zk-SNARK when $T > 460$ for 16-bit vnTinyRAM, and when $T > 326$ for 32-bit vnTinyRAM; moreover, these savings in space grow unbounded as T increases.

Finally, being scalable, our zk-SNARK implementation is the first to achieve a well-defined clock rate of *verified instructions per second* (VIPS). Concretely, for vnTinyRAM, we obtain the following VIPS values:

	16-bit vnTinyRAM $(w,k) = (16,16)$	32-bit vnTinyRAM $(w,k) = (32,16)$
1 thread	VIPS $= \frac{1}{33.1}$Hz	VIPS $= \frac{1}{35.5}$Hz
4 threads	VIPS $= \frac{1}{11.5}$Hz	VIPS $= \frac{1}{12.1}$Hz

While perhaps too slow for most applications, our prototype empirically demonstrates the feasibility of the bootstrapping approach as a way to achieve scalability of zk-SNARKs and, more generally, to achieve the rich functionality of proof-carrying data.

Acknowledgments. We thank Andrew V. Sutherland for generous help in running the CM method on elliptic curves with large discriminants. We thank Damien Stehlé and Daniele Micciancio for discussions about the security of subset-sum functions. We thank Koray Karabina for answering questions about algorithms in [KT08].

This work was supported by: the Broadcom Foundation and Tel Aviv University Authentication Initiative; the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370; the Check Point Institute for Information Security; the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 240258; the Israeli Centers of Research Excellence I-CORE program (center 4/11); the Israeli Ministry of Science and Technology; the Leona M. & Harry B. Helmsley Charitable Trust; the Simons Foundation, with a Simons Award for Graduate Students in Theoretical Computer Science; and the Skolkovo Foundation with agreement dated 10/26/2011.

References

[Ajt96] Ajtai, M.: Generating hard instances of lattice problems. In: STOC 1996 (1996)

[AM93] Atkin, A.O.L., Morain, F.: Elliptic curves and primality proving. Math. Comp (1993)

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

[BCG+13a] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: Verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 90–108. Springer, Heidelberg (2013)

[BCG+13b] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: TinyRAM architecture specification v2.00 (2013), URL: `http://scipr-lab.org/tinyram`

[BCG+14] Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from Bitcoin. In: SP 2014 (2014)

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

[BCTV14a] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. Cryptology ePrint Archive (2014)

[BCTV14b] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von Neumann architecture. In: Security 2014 (2014), `http://eprint.iacr.org/2013/879`

[BDSMP91] Blum, M., De Santis, A., Micali, S., Persiano, G.: Non-interactive zero-knowledge. SIAM J. Comp. (1991)

[BEG+91] Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: FOCS 1991 (1991)

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

[BFM88] Blum, M., Feldman, P., Micali Non-interactive, S.: zero-knowledge and its applications. In: STOC 1988 (1988)

[BFR+13] Braun, B., Feldman, A.J., Ren, Z., Setty, S., Blumberg, A.J., Walfish, M.: Verifying computations with state. In: SOSP 2013 (2013)

[BS10] Benger, N., Scott, M.: Constructing tower extensions of finite fields for implementation of pairing-based cryptography. In: Hasan, M.A., Helleseth, T. (eds.) WAIFI 2010. LNCS, vol. 6087, pp. 180–195. Springer, Heidelberg (2010)

[BSW12] Boneh, D., Segev, G., Waters, B.: Targeted malleability: Homomorphic encryption for restricted computations. In: ITCS 2012 (2012)

[CMT12] Cormode, G., Mitzenmacher, M., Thaler, J.: Practical verified computation with streaming interactive proofs. In: ITCS 2012 (2012)

[CRR11] Canetti, R., Riva, B., Rothblum, G.N.: Practical delegation of computation using multiple servers. In: CCS 2011 (2011)

[CT10] Chiesa, A., Tromer, E.: Proof-carrying data and hearsay arguments from signature cards. In: ICS 2010 (2010)

[CT12] Chiesa, A., Tromer, E.: Proof-carrying data: Secure computation on untrusted platforms (high-level description). In: The Next Wave: The National Security Agency's Review of Emerging Technologies (2012)

[CTV13] Chong, S., Tromer, E., Vaughan, J.A.: Enforcing language semantics using proof-carrying data. ePrint 2013/513 (2013)

[ES10] Enge, A., Sutherland, A.V.: Class invariants by the CRT method. In: Hanrot, G., Morain, F., Thomé, E. (eds.) ANTS-IX. LNCS, vol. 6197, pp. 142–156. Springer, Heidelberg (2010)

[FST10] Freeman, D., Scott, M., Teske, E.: A taxonomy of pairing-friendly elliptic curves. Journal of Cryptology (2010)

[GGH96] Goldreich, O., Goldwasser, S., Halevi, S.: Collision-free hashing from lattice problems. Technical report, ECCC TR95-042 (1996)

[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)

[Gro10] Groth, J.: Short pairing-based non-interactive zero-knowledge arguments. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 321–340. Springer, Heidelberg (2010)

[KT08] Karabina, K., Teske, E.: On prime-order elliptic curves with embedding degrees $k = 3$, 4, and 6. In: van der Poorten, A.J., Stein, A. (eds.) ANTS-VIII 2008. LNCS, vol. 5011, pp. 102–117. Springer, Heidelberg (2008)

[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)

[Lip13] Lipmaa, H.: Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part I. LNCS, vol. 8269, pp. 41–60. Springer, Heidelberg (2013)

[LMN10] Lauter, K., Montgomery, P.L., Naehrig, M.: An analysis of affine coordinates for pairing computation. In: Joye, M., Miyaji, A., Otsuka, A. (eds.) Pairing 2010. LNCS, vol. 6487, pp. 1–20. Springer, Heidelberg (2010)

[Mic00] Micali, S.: Computationally sound proofs. SIAM J. Comp. (2000)

[MNT01] Miyaji, A., Nakabayashi, M., Takano, S.: New explicit conditions of elliptic curve traces for FR-reduction. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences (2001)

[NY90] Naor, M., Yung, M.: Public-key cryptosystems provably secure against chosen ciphertext attacks. In: STOC 1990 (1990)

[PGHR13] Parno, B., Gentry, C., Howell, J., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: Oakland 2013 (2013)

[SBV+13] Setty, S., Braun, B., Vu, V., Blumberg, A.J., Parno, B., Walfish, M.: Resolving the conflict between generality and plausibility in verified computation. In: EuroSys 2013 (2013)

[SBW11] Setty, S., Blumberg, A.J., Walfish, M.: Toward practical and unconditional verification of remote computations. In: HotOS 2011 (2011)

[SMBW12] Setty, S., McPherson, M., Blumberg, A.J., Walfish, M.: Making argument systems for outsourced computation practical (sometimes). In: NDSS 2012 (2012)

[Sut11] Sutherland, A.V.: Computing Hilbert class polynomials with the Chinese remainder theorem. Math. Comp. (2011)

[Sut12] Sutherland, A.V.: Accelerating the CM method. LMS Journal of Computation and Mathematics (2012)

[SVP+12] Setty, S., Vu, V., Panpalia, N., Braun, B., Blumberg, A.J., Walfish, M.: Taking proof-based verified computation a few steps closer to practicality. In: Security 2012 (2012)

[Tha13] Thaler, J.: Time-optimal interactive proofs for circuit evaluation. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 71–89. Springer, Heidelberg (2013)

[TRMP12] Thaler, J., Roberts, M., Mitzenmacher, M., Pfister, H.: Verifiable computation with massively parallel interactive proofs. CoRR (2012)

[Val08] Valiant, P.: Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In: Canetti, R. (ed.) TCC 2008. LNCS, vol. 4948, pp. 1–18. Springer, Heidelberg (2008)