POSTER: Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture

Eli Ben-Sasson*, Alessandro Chiesa†, Eran Tromer‡, Madars Virza†
*Technion, eli@cs.technion.ac.il
†MIT, {alexch, madars}@mit.edu
‡Tel Aviv University, tromer@cs.tau.ac.il

***Abstract*—We build a system that can prove, in zero knowledge, the correct executions of programs on a 32-bit RISC machine. The proofs are succinct: short and easy to verify.**

All previous implementations of zero-knowledge proof systems with succinct proofs (also known as *zk-SNARK*s) require the proof system's keys to be regenerated for each distinct program; this is costly and requires the intervention of a trusted party. Our system is universal: no program-specific setup is required.

I. Introduction

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

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

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

- The proof system's *soundness* guarantees that, if the NP statement is false, the prover cannot convince the verifier. Thus, soundness addresses the client's integrity concern.
- *Zero-knowledge* guarantees that, if the NP statement is true, the prover can convince the verifier without leaking information about w; this addresses the server's confidentiality.

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

Efficiency. Besides the aforementioned security desiderata, many settings also call for *efficiency* desiderata. The client may be either unable or unwilling to engage in lengthy interactions with the server, or to perform large computations beyond the "bare minimum" of sending the input x and receiving the output z. For instance, the client may be a computationally-weak device with intermittent connectivity (e.g., a smartphone).

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

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

Many works have obtained zk-SNARK constructions. Three of these [1, 2, 3] provide implementations, but none of them work for a universal machine. Instead, the proof system needs to be setup anew for each separate program F; each such setup is both expensive and requires a trusted party.

Outsourcing computation to powerful servers. Numerous works seek to verifiably outsource computation to untrusted powerful servers. Verifiable outsourcing of computations *is not our goal*. Rather, we study *non-interactive zero-knowledge proofs*, which are useful even when applied to relatively-small computations, and even with high overheads.

II. Contributions

We obtain a zk-SNARK that supports *executions on a universal von Neumann RISC machine*. Our zk-SNARK consists of two components: a new circuit generator and a new zk-SNARK for circuits. These can be used independently, or combined to obtain an overall system.

A. A new circuit generator

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

(1) Our circuit generator is *universal*: when given input bounds ℓ, n, T, it produces a circuit that can verify the execution of *any* program with $\leq \ell$ instructions, on *any* input of size $\leq n$, for $\leq T$ steps. Instead, all prior circuit generators [4, 5, 1, 2, 3] hardcoded the program in the circuit. Combined with a zk-SNARK for circuits (or any NP proof system for circuits),

we achieve a notable conceptual advance: *once-and-for-all key generation* that allows verifying all programs up to a given size. This removes major issues in all prior systems: expensive per-program key generation, and the thorny issue of conducting it anew in a trusted way for every program.

Our circuit generator supports a universal machine that, like modern computers, follows the *von Neumann paradigm* (program and data lie in the same read/write address space). Concretely, it supports a von Neumann RISC architecture called vnTinyRAM, a modification of TinyRAM [2]. Thus, we also support programs leveraging techniques such as *just-in-time compilation*. (To compile C programs to vnTinyRAM, we ported the GCC compiler, building on the work of [2].)

See Figure 1 for a functionality comparison with prior circuit generators (for details, see [3, §2]).

Supported functionality	[4, 5, 1]	[2]	[3]	**this work**
side-effect free comp.	✓	✓	✓	✓
data-dep. mem. accesses	×	✓	✓	✓
data-dep. contr. flow	×	✓	×	✓
self-modifying code	×	×	×	✓
universality	×	×	×	✓

Fig. 1: Functionality comparison among circuit generators.

(2) Our circuit generator handles *larger* arbitrary programs: the size of the circuit $C_{\ell,n,T}$ is $O\big((\ell+n+T)\cdot\log(\ell+n+T)\big)$ gates. Thus, the dependence on program size is *additive*, instead of multiplicative as in [2], where the generated (non-universal) circuit has size $\Theta\big((n+T)\cdot(\log(n+T)+\ell)\big)$. As Figure 2 shows, our efficiency improvement compared to [2] is not merely asymptotic but yields sizable concrete savings.

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

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

An efficiency comparison with other non-universal circuit generators [4, 5, 1, 3] varies from program to program. We find via experiment that such circuit generators perform better than ours for programs that are "close to a circuit", and worse for those rich in data-dependent memory accesses and control flow.

B. A new zk-SNARK for circuits

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

(3) We improve upon and implement the protocol of Parno et al. [1]. Unlike previous zk-SNARK implementations [1, 2, 3], we do not use off-the-shelf cryptographic libraries. Rather, we create a tailored implementation of the requisite components.

To facilitate comparison with prior work, we instantiate our techniques for two specific algebraic setups, one based on Edwards curves at 80 bits of security (as in [2]), and one on Barreto–Naehrig curves at 128 bits of security (as in [1, 3]).

On a typical desktop, proof verification is fast: at 80-bit security, for an n-byte input to the circuit, verification takes $4.7 + 0.0004 \cdot n$ milliseconds, *regardless of circuit size*; at 128-bit security, it takes $4.8 + 0.0005 \cdot n$. Key generation and proof generation entail a per-gate cost. E.g., for a circuit with 16 million gates: at 80 bits of security, key generation takes 81 µs per gate and proving takes 109 µs per gate; at 128 bits of security, these per-gate costs increase to 100 µs and 144 µs.

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

Compared to previous implementations of zk-SNARKs for circuits [1, 2, 3], our implementation improves both proving and verification times, e.g., see Figure 3.

	80 bits of security			128 bits of security		
	[2]	**this**	improv.	[1]	**this**	improv.
Key gen.	306 s	97 s	3.2×	123 s	117 s	1.1×
Prover	351 s	115 s	3.1×	784 s	147 s	5.3×
Verifier	66.1 ms	4.9 ms	13.5×	9.2 ms	5.1 ms	1.8×
Proof size	322 B	230 B	1.4×	288 B	288 B	(same)

Fig. 3: Comparison with prior zk-SNARKs for a 1-million-gate arithmetic circuit and a 1000-bit input, running on our benchmarking machine.

C. Two components: independent or combined

Our circuit generator and zk-SNARK for circuits can be used independently, or combined. When combined, we obtain a *zk-SNARK for proving/verifying correctness of* vnTinyRAM *computations*; see Figure 4 and Figure 5 for diagrams of this system. We evaluated this overall system for programs with up to 10,000 instructions, running for up to 32,000 steps.

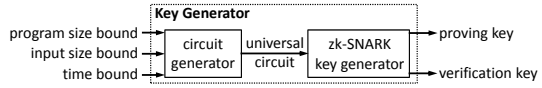

Fig. 4: **Offline phase (once).** The key generator outputs proving and verification keys, for proving/verifying correctness of suitably-bounded program executions.

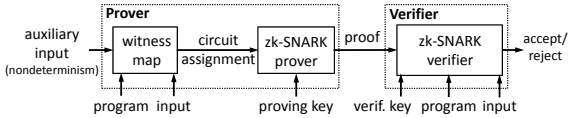

Fig. 5: **Online phase (any number of times).** The prover sends a short and easy-to-verify proof to a verifier. This can be repeated any number of times.

[1] B. Parno, C. Gentry, J. Howell, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *Oakland '13*.

[2] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *CRYPTO '13*.

[3] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in *SOSP '13*.

[4] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish, "Taking proof-based verified computation a few steps closer to practicality," in *Security '12*.

[5] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish, "Resolving the conflict between generality and plausibility in verified computation," in *EuroSys '13*.