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

by

Madars Virza
B.Sc., University of Latvia (2011)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 2014

© Massachusetts Institute of Technology 2014. All rights reserved.
SNARKs for C: Verifying Program Executions
Succinctly and in Zero Knowledge
by
Madars Virza

Submitted to the
Department of Electrical Engineering and Computer Science
on September 16, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

We present a proof system that allows efficient verification of NP statements, given proofs produced by an untrusted yet computationally-bounded prover. Our system is publicly verifiable: after a trusted third-party has generated a proving key and a verification key, anyone can use the proving key to generate non-interactive proofs for adaptively-chosen NP statements, and the proofs can be verified by anyone using the verification key. Moreover, our system is statistically zero-knowledge and the generated public parameters are reusable.

The NP-complete language we choose is the correct execution of programs on TinyRAM, a minimalistic (nondeterministic) random-access machine that we design. Together with TinyRAM port of gcc compiler this achieves the first practical realization of a zero-knowledge Succinct Non-interactive ARgument of Knowledge (zk-SNARK) for program executions, in the preprocessing model. This cryptographic primitive is a powerful solution for delegating NP computations, and enjoys many features not achieved by primitives implemented in prior works, most importantly, succinct verification and support for arbitrary computations.

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

1. A transformation that, given as input a C program, outputs a circuit whose satisfiability encodes the correct execution of the program. We leverage nondeterminism to make the generated circuit’s size merely quasilinear in the size of the computation; in particular, we efficiently handle arbitrary loops, control flow, and random-memory accesses. This is in contrast with existing “circuit compilers”, which produce circuits of quadratic size.

2. A transformation that, given as input a linear PCP for verifying satisfiability of circuits, outputs a corresponding SNARK. Furthermore, by building on recent work about quadratic span programs, using suitable choices of finite field and FFT algorithms, we give a very efficient implementation of a zero-knowledge linear PCP: linear-time query generation and quasilinear-time prover.

Thesis Supervisor: Ronald L. Rivest
Title: Andrew and Erna Viterbi Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Eli Ben-Sasson
Title: Associate Professor of Computer Science, Technion – Israel Institute of Technology
Acknowledgments

First, I would like to thank Ron Rivest for his direction and guidance, insightful advice and the many opportunities I have had. His wealth of knowledge and well-timed encouragement have been tremendously helpful for both shaping this research and my graduate studies in general.

Additionally, I would like to thank Alessandro Chiesa, for introducing me to succinct verification, always being available to share his knowledge and experience, and above all: for being a great friend.

I would also like to thank Eli Ben-Sasson for many fascinating discussions about PCPs and related algebraic problems and his invitation to expand this research for the benefit of practical PCPs. Eli also invited me to spend a summer in Technion, which accelerated progress on many other fronts, for which I am grateful.

I would also like to thank Eran Tromer for providing me with lots of constructive feedback and technical advice and always rising insightful counterpoints in our discussions. Eran also hosted me in Israel and patiently listened to endless number of practice talks, which helped our presentation a lot.

This research has also benefited from many fruitful discussions with Nickolai Zeldovich, whose suggestions and comments helped improve this research greatly.

This thesis is largely based on the work I did jointly with Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin and Eran Tromer and that appeared in our paper “SNARKs for C: Verifying Computations Succinctly and Zero Knowledge” [BCGTV13a]. I am grateful to Eli, Ale, Daniel and Eran for the fruitful collaboration.

The authors of the paper gratefully thank the members of the programming team: Ohad Barta, Lior Greenblatt, Shaul Kfir, Lev Pachmanov, Michael Riabzev, Gil Timnat, and Arnon Yogev. The authors of the paper also thank Dan Berstein, Tanja Lange, Peter Schwabe, and Andrew Sutherland for discussions about elliptic curves; and Andrey Vihrov for helpful technical discussions. We would also like to thank Nickolai Zeldovich for allowing us to use servers of his research group, on which all experiments reported here were carried out.

Finally, I would like to thank my family and my friends both here in the US and back home.

The research leading to these results, and in particular 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 also 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).
## Contents

1 Introduction ................................................. 9
   1.1 Overview ........................................ 9
   1.2 Motivating examples .............................. 10
   1.3 Contributions .................................. 11
   1.4 Bibliographic notes ............................ 14

2 Preliminaries .............................................. 15
   2.1 Succinct Verification in the Preprocessing Model .... 15
   2.2 Our approach ................................... 16
   2.3 System overview ................................ 17

3 Verifying Circuit Satisfiability via Linear PCPs ........ 19
   3.1 Definition of Zero-Knowledge SNARKs for Circuit Satisfiability .... 19
   3.2 Definition of Linear PCPs .......................... 21
   3.3 Constructing a SNARKs from Linear PCPs ............ 23
   3.4 An Efficient Linear PCP ............................ 24
   3.5 Optimizing the transformation from Linear PCPs to SNARKs .......... 32

4 A New Quasi-linear Circuit Generator for RAM Machines .... 37
   4.1 The TinyRAM Architecture ....................... 37
   4.2 A Compiler from C to TinyRAM .................... 39
   4.3 An Efficient Reduction from TinyRAM to Circuit Satisfiability .... 43

5 Design of algebraic time and memory consistency checkers .... 47

6 Experimental evaluation ..................................... 53
   6.1 Instantiation of $G_1, G_2, G_T$ and pairing ............. 53
   6.2 Performance of our zk-SNARK for circuit satisfiability .... 54
   6.3 Performance of our zk-SNARK for arbitrary C programs .... 57
   6.4 Future optimizations ................................ 62

7 Conclusion .................................................. 65
   7.1 Future work .................................... 65

A Examples Used in Section 4.2 ................................ 67
Chapter 1

Introduction

1.1 Overview

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 multiparty computation [GMW87, BGW88], group signatures [BW06, Gro06], malleable proof systems [CKLM12], anonymous credentials [BCKL08], delegatable credentials [BCCK+09], electronic voting [KMO01, Gro05, Lip11], and many others. Known constructions of zero-knowledge proofs of knowledge are practical only when proving statements of certain form, which 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 [BBKSS09, ABBB+12].

Due to differences in computational power among parties, many applications (e.g., when outsourcing computation to an untrusted worker) 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 (and more efficient) to express via algorithms written in a high-level programming language. Modern compilers can efficiently transform these algorithms into a program to be executed on a random-access machine (RAM) [CR72, AV77]. Therefore, we seek proof systems that efficiently support NP statements expressed as the correct execution of a RAM program.

In this work we present an implementation of a publicly-verifiable non-interactive ar-
argument system for \textit{NP}. The system, moreover, is a zero-knowledge proof-of-knowledge. It directly proves correct executions of programs on \textit{TinyRAM}, a random-access machine tailored for efficient verification of nondeterministic computations.

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 \textit{TinyRAM} port of the \texttt{gcc} compiler.

This yields a zero-knowledge Succinct Non-interactive ARgument of Knowledge (shortened as “zk-SNARK”) for program executions in the preprocessing model — a powerful solution for delegating \textit{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 on and improving on recent work on quadratic arithmetic programs.

1.2 Motivating examples

Ability to efficiently prove arbitrary \textit{NP} statements has immediate applicability to outsourcing computation. That is, a computationally weak client can give his long-running computations to a third party, without worrying about its integrity as the results would be accompanied by easy-to-verify mathematical proof. We now proceed to give two other concrete examples, where our system could be useful:

\textbf{Distributed computation in adversarial setting.} Having zk-SNARKs also solves a much harder problem of ensuring correctness of distributed computations among mutually untrusting parties, by instantiating an approach called “proof-carrying-data” [CT10]. There the system designer would express properties of computation’s outputs as an \textit{NP} statement. All messages exchanged would be accompanied by zero-knowledge proofs that attest that message and all its history has complied with the specific properties.

\textbf{Computing on data you don’t own.} We also believe that efficient zero knowledge proofs have potential to transform biomedical computations. For example, companies like 23andMe [23a] maintain genome databases that could be very useful for advancing our understanding of health, but cannot be shared with the larger research community due to privacy concerns.
A potential solution could be allowing researchers to give their programs to the holders of the databases, which the database holders would (for a fee) execute and give back the corresponding results. This poses conflicting interests:

- database owners have financial incentives to not execute long-running simulations, but just make up plausibly-looking results. Catching this would pose a serious problem, as the input to the program — the database itself — is kept private.

- database owners could attach mathematical proofs of correct execution of researcher’s programs, but what if the proof leaks more than just the claimed result, e.g. someone’s entire genome?

The proof system described in our work achieves informational theoretic zero-knowledge properties, thereby even unbounded attacker cannot learn anything more from it than the claimed result. Therefore such proofs are safe to attach, enabling a new way to conduct research.

### 1.3 Contributions

To attain practical SNARKs for arbitrary programs the contribution of this thesis is two-fold. First, we bring numerous improvements to existing theoretical work so that when implemented our algorithms will enjoy good concrete efficiency. Second, we present a working prototype that, to best of our knowledge, is the first real-world realization of succinct verification for arbitrary programs. 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 essentially-optimal asymptotic efficiency: linear-time generator, quasilinear-time prover, and linear-time verifier.

   Our approach consists of two steps. First, we optimized and implemented the transformation of Bitansky et al. [BCIOP13]; our optimizations rely on multi-exponentiation algorithms (see [Ber02] and references therein) and on a specialized choice of elliptic curve. 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.

   When used in the aforementioned transformation, our linear PCP yields a zero-knowledge SNARK with essentially optimal asymptotic efficiency and whose proofs are only 2576 bits long (independent of $C$ and $x$).

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. Also delegation schemes presented by [Gro10b, Lip12, GGPR13, BCIOP13] assume that the computation that needs to be verified is represented as an 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.

Compilers do a great job in reducing a real program written in some high level programming language like C to a low level programing language like the assembly language of some CPU. Furthermore, we observe that it is usually possible to convert the assembly language of some CPU to an assembly language of some other CPU in linear time with small hidden constants. Thus, we don’t need ppSNARKS for C programs but only for some assembly programs. Therefore, following the theoretical work of Ben-Sasson et al. [BCGT13a], we additionally (1) design a minimalistic (nondeterministic) random-access machine with a Harvard architecture that we call TinyRAM, and (2) implement a transformation from a TinyRAM program (and a time bound) to a corresponding circuit. We complement the transformation with a gcc backend, for compiling C programs into TinyRAM assembly.

Our reduction is qualitatively different from all previous implementations of “circuit generators” (e.g., Fairplay [MNPS04, BDNP08]): it leverages nondeterminism to significantly 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 implementation produces circuits of only $O(T(\log T)^2)$ size.

Our choice of architecture for TinyRAM strikes a balance between allowing for efficient compilation of programs into assembly code, and the need to design small circuits for verifying correctness of the transition function of the machine.

Delegation for NP programs. Combined, our contributions yield a system for verifying 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. See Figure 1-1 for a high-level system overview.

Reusable components. Our contributions are independent in that each can be useful without the others:

- If one designed a linear PCP for circuits that is more efficient than ours, it could be plugged into our transformation to SNARKs. Moreover, such a linear PCP would also benefit from our circuit generator for TinyRAM programs, and our compiler from C programs to TinyRAM assembly.

- If one had an NP problem already represented via arithmetic circuit satisfiability (for instance, this is simple to achieve when considering “structured” computational problems such as evaluating FFTs) then there is no need to reduce from C (or TinyRAM) programs, so one could directly invoke our zk-SNARK.

- Our reduction from C programs to circuit satisfiability can be used in conjunction with other proof systems built for circuit satisfiability (or other related algebraic satisfaction problems). For instance, it can be used with many recent constructions of non-interactive zero-knowledge proofs [GOS06a, GOS06b, AF07, Gro09, Gro10a, Gro10b].
On the left: a C program can be compiled into a corresponding TinyRAM program.

**Offline phase:** the key generator computes a (long) proving key and a (short) verification key for proving/checking correct (nondeterministic) computations of a given TinyRAM program (for at most a given number of time steps on inputs of a given size); our compiler can be used to obtain TinyRAM programs. Without loss of generality, we consider the language of accepting computations without output. This can be always guarantee as the output of the computation can be treated as input for the decider of the same computation that checks if the output matches the reference output provided as decider’s input.

**Online phase:** the prover sends a non-interactive publicly-verifiable proof to a verifier; this phase can be repeated any number of times.

Figure 1-1: High-level overview of our zk-SNARK system for arbitrary C programs.
1.4 Bibliographic notes

The research described in this thesis was done jointly with Professor Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin and Professor Eran Tromer. The author’s contributions lie primarily in the following areas: helping to design the algebraic time and memory consistency checkers, helping to improve the efficiency of the circuit generator and LPCP prover, and helping to implement the SNARK for TinyRAM. The write-up in this thesis is based on the extended version of our jointly authored paper “SNARKs for C: Verifying Computations Succinctly and Zero Knowledge” [BCGTV13a], a conference version of which appeared in the proceedings of the 33rd Annual International Cryptology Conference (CRYPTO 2013) in August 2013.

The rest of this thesis is organized as follows. In Chapter 2 we give a precise definition of our goal, describe our approach and motivate our choices, contrasting them with previous work. In Chapter 3 we define linear PCPs, how to use them to obtain SNARKs for the circuit satisfiability problem and describe our Linear PCP and our transformation from LPCPs to SNARKs. In Chapter 4 we describe TinyRAM, our architecture for fast verification of programs, our compiler from C to TinyRAM and describe our circuit generator in detail. Small algebraic time and memory consistency checkers form the core part of our circuit reduction and we devote Chapter 5 to describing them. Finally, we describe our experimental setup and end-to-end system evaluations in and in Chapter 7 we conclude.
Chapter 2

Preliminaries

2.1 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.

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. Computational soundness seems inherent in many of these cases [BH87, GH98, GV02, Wee05]. Proofs (whether interactive or not) that are only computationally sound are also known as arguments [BCC88].

In this work we seek non-interactive succinct verification. We make two concessions: soundness is computational, and the proofs are in the preprocessing model which relies on an expensive but reusable key generation (discussed below). Thus, we investigate efficient implementations of succinct non-interactive arguments (SNARGs) in the preprocessing model. 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.

Definition 1. The Boolean circuit satisfaction problem of a Boolean circuit \( C : \{0, 1\}^n \times \{0, 1\}^h \rightarrow \{0, 1\} \) is the relation \( \mathcal{R}_C = \{(x, w) \in \{0, 1\}^n \times \{0, 1\}^h : C(x, w) = 1\} \); its language is \( \mathcal{L}_C = \{x \in \{0, 1\}^n : \forall w \in \{0, 1\}^h C(x, w) = 1\} \). For a family of Boolean circuits

\[
\mathcal{C} = \{C_\ell : \{0, 1\}^{n(\ell)} \times \{0, 1\}^{h(\ell)} \rightarrow \{0, 1\}\}_{\ell \in \mathbb{N}},
\]

we denote the corresponding infinite relation and language by \( \mathcal{R}_C = \bigcup_{\ell \in \mathbb{N}} \mathcal{R}_{C_\ell} \) and \( \mathcal{L}_C = \bigcup_{\ell \in \mathbb{N}} \mathcal{L}_{C_\ell} \).

A publicly-verifiable preprocessing SNARG (or, simply SNARG) is a triple of algorithms \((G, P, V)\), respectively called key generator, prover, and verifier, working as follows. The (probabilistic) key generator \( G \), given a security parameter \( \lambda \) and circuit \( C : \{0, 1\}^n \times \{0, 1\}^h \rightarrow \{0, 1\} \), outputs a proving key \( \sigma \) and a verification key \( \tau \); these are the system’s public parameters, which need to be generated only once per circuit. After
that, anyone who can determine the necessary witness \( w \) can use the proving key \( \sigma \) to generate non-interactive proofs for the language \( \mathcal{L}_C \), and anyone can use the verification key \( \tau \) to check these proofs. Namely, given \( \sigma \) and any \((x, w) \in \mathcal{R}_C\), the honest prover \( P(\sigma, x, w) \) produces a proof \( \pi \) attesting that \( x \in \mathcal{L}_C \); the verifier \( V(\tau, x, \pi) \) checks that \( \pi \) 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 \( \text{poly}(|C|) \) operations;
- running the prover \( P \) on input \((\sigma, x, a)\) requires \( \text{poly}(|C|) \) operations; but
- running the verifier \( V \) on input \((\tau, x, \pi)\) requires only \( \text{poly}(|x|) \) operations; and
- an honestly-generated proof has size \( \text{poly}(\lambda) \).

We require (adaptive) computational soundness: for every polynomial-size prover \( P^* \), constant \( c > 0 \), large enough security parameter \( \lambda \in \mathbb{N} \), and circuit \( C : \{0, 1\}^n \times \{0, 1\}^h \rightarrow \{0, 1\} \) of size \( \lambda^c \), letting \((\sigma, \tau) \leftarrow G(1^\lambda, C)\), if \( P^*(\sigma, \tau) \) outputs an adaptively-chosen \((x, \pi)\) 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):

\[
\Pr \left[ V(\tau, x, \pi) = 1 \mid \nexists w \text{ s.t. } (x, w) \in \mathcal{R}_C \right. 
\left( (\sigma, \tau) \leftarrow G(1^\lambda, C) \right. 
\left. (x, \pi) \leftarrow P^*(\sigma, \tau) \right) \leq \text{negl}(\lambda) .
\]

If a SNARG satisfies a certain natural proof-of-knowledge property, we call it a SNARG of knowledge (SNARK). If it also satisfies a certain natural zero-knowledge property, we call it a zero-knowledge SNARK (zk-SNARK).

### 2.2 Our approach

It would be wonderful to have efficient and generic implementations of SNARGs without any expensive preprocessing. (I.e., have the generator \( G \) run in \( \text{poly}(\lambda) \) instead of \( \text{poly}(|C|) \) cryptographic operations). The two known approaches to constructing such SNARGs are Micali’s “computationally-sound proofs” \([\text{Mic00}]\), and the bootstrapping techniques of Bitansky et al. \([\text{BCCT13}]\). Algorithmically, both are complex constructions: the former requires probabilistically-checkable proofs (PCPs) \([\text{BFLS91}]\) (which remain concretely expensive despite recent advances \([\text{BGHSV05}, \text{BS08}, \text{Din07}, \text{MR08}, \text{BCGT13b}]\)), and the latter uses recursive proof-composition which adds a (quasilinear yet) concretely large overhead.\(^\dagger\)

Thus, 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

\(^\dagger\)Moreover, giving up public verifiability does not seem to allow for significantly simpler constructions. Concretely, known constructions of privately-verifiable SNARGs (without preprocessing) \([\text{BCCT12}, \text{DFH12}, \text{GLR11}, \text{BC12}]\) rely, not only on PCPs, but also on private-information retrieval or fully-homomorphic encryption, both of which are expensive in practice.
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 [Gro10a, Lip12, GGPR13, BCIOP13] 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. [BCIOP13] gave a general technique to construct 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\(^2\). 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.

Efficient HVZK linear PCPs for circuit satisfiability, with low-degree verifiers, are implied by the work of Gennaro et al. [GGPR13] on quadratic-span programs (QSPs) and quadratic arithmetic programs (QAPs). Moreover, the work of Ben-Sasson et al. [BCGT13a] implies that random-access machine computations can be efficiently reduced to circuit satisfiability. Combining these ingredients, one obtains a theoretically simple and attractive route for constructing zk-SNARKs. As always, bringing theory to practice requires significant additional insights and improvements, and tackling these is the goal of our work.

### 2.3 System overview

**RAM machine and compiler.** Our ultimate goal is to obtain a delegation schemes for real life programs written in some high level programming language such as C. In order to achieve this goal we have to also implement a compiler that will be able to convert a programs written in some high level programming language such as C to a specification of a RAM. Thus, we first have to define a RAM architecture that has two main properties. First, the resulting circuit obtained from invoking the work of [BCGT13a] on this RAM are as small as possible. Second, it is possible to efficiently compile high level programming language such as C to this architecture. We call this architecture TinyRAM (see Section 4.1 for a discussion about the TinyRAM architecture).

**RAM to circuits reduction.** With this architecture in hand, we can outline our solution for obtaining the first practical constructions of a delegation of computation system for C programs. First, we would like to compile C programs into TinyRAM assembly. Second, by running the TinyRAM assembly code, using a TinyRAM simulator, we will obtain a computation transcript of the TinyRAM machine. Third, we will reduce the problem of verifying this transcript to a circuit satisfiability problem (CSAT).

---

\(^2\)More precisely, the soundness is required to hold only against affine proof oracles (i.e. oracles $O_{II,b}$ that on input $x$ produce $IIx + b$ for some matrix $II$ and vector $b$), but this difference does not significantly alter the proofs or later use of the approach.
Again, we refer reader to Figure 1-1 for a high-level system overview.
Chapter 3

Verifying Circuit Satisfiability via Linear PCPs

Our high-level approach to obtain the zk-SNARK for verifying circuit satisfiability is as follows.

- First, we optimized and implemented the transformation of Bitansky et al. [BCIOP13]; the transformation takes as input any honest-verifier zero-knowledge (HVZK) linear PCP and outputs a zk-SNARK.
- Second, we provide an efficient implementation of a HVZK linear PCP for circuit satisfiability.

This chapter is organized as follows. We first give the necessary definitions and survey the prior work: in Section 3.1 we give formal definitions of zero-knowledge SNARKs for Circuit Satisfiability problem and in Section 3.2 we give formal definition of linear PCPs; and in Section 3.3 we discuss the transformation from a linear PCP to a zk-SNARK.

Then we proceed to describe our contributions: in Section 3.4 we discuss our linear PCP and in Section 3.5 we discuss our improvements to the resulting SNARK.

3.1 Definition of Zero-Knowledge SNARKs for Circuit Satisfiability

In Section 2.1 we informally introduced (publicly-verifiable preprocessing) zero-knowledge SNARKs for Boolean circuit satisfiability. For completeness, we give here formal definitions. The extension of the definition to arithmetic circuit satisfiability (as defined in Section 3.4) is straightforward, so we omit it.\(^1\)

As defined below our SNARK prover will have as its arguments not only the input \(x\) and witness \(w\), but also the assignment \(a\) of all wire values of the circuit. However, both arithmetic and Boolean circuits are deterministic models of computation so \((x, w)\) uniquely determines \(a\) and (as \((x, w)\) is part of \(a\)), so we will assume that our SNARK prover knows

\(^1\)In particular, we do not define here zero-knowledge SNARKs relative to a universal relation [BG08]. For details, see [BCIOP13].
the entire assignment $a$ (and does need to recompute it from $(x, w)$) and will use those terms interchangeably.

**Definition 3.1.1.** A triple of algorithms $(G, P, V)$ is a publicly-verifiable preprocessing SNARG (or, simply SNARG) for Boolean circuit satisfiability if the following conditions are satisfied.

1. **Completeness** For every sufficiently large security parameter $\lambda \in \mathbb{N}$, every circuit $C : \{0, 1\}^n \times \{0, 1\}^h \rightarrow \{0, 1\}$, every input $x \in \{0, 1\}^n$, and every assignment $a \in \{0, 1\}^h$ with $(x, a) \in \mathcal{R}_C$,

   $$\Pr \left[ V(\tau, x, \pi) = 1 \mid (\sigma, \tau) \leftarrow G(1^\lambda, C), \pi \leftarrow P(\sigma, x, a) \right] = 1 .$$

2. **Soundness** For every polynomial-size prover $P^*$, constant $c > 0$, every large enough security parameter $\lambda \in \mathbb{N}$, and every circuit $C : \{0, 1\}^n \times \{0, 1\}^h \rightarrow \{0, 1\}$ of size $\lambda^c$,

   $$\Pr \left[ V(\tau, x, \pi) = 1 \mid \hat{\#} a \text{ s.t. } (x, a) \in \mathcal{R}_C, (\sigma, \tau) \leftarrow G(1^\lambda, C), (x, \pi) \leftarrow P^*(\sigma, \tau) \right] \leq \negl(\lambda) .$$

3. **Efficiency** There is a universal polynomial $p$ such that, for every large enough security parameter $\lambda \in \mathbb{N}$, every circuit $C : \{0, 1\}^n \times \{0, 1\}^h \rightarrow \{0, 1\}$, input $x \in \{0, 1\}^n$, and assignment $a \in \{0, 1\}^h$ with $(x, a) \in \mathcal{R}_C$,

   - the key generator $G$ runs in time $p(\lambda + |C|)$;
   - the prover $P$ runs in time $p(\lambda + |C|)$;
   - the verifier $V$ runs in time $p(\lambda + |x|)$;
   - an honestly generated proof has size $p(\lambda)$.

**Proof of knowledge.** A SNARG of knowledge (SNARK) is a SNARG where soundness is strengthened as follows:

**Definition 3.1.2.** A triple of algorithms $(G, P, V)$ is a SNARK (for circuit satisfiability) if it is a SNARG (for circuit satisfiability) where soundness is replaced by the following stronger requirement:
Proof of knowledge

For every polynomial-size prover $P^*$ there exists a polynomial-size extractor $E$ such that for every constant $c > 0$, large enough security parameter $\lambda \in \mathbb{N}$, every auxiliary input $z \in \{0,1\}^{\text{poly}(\lambda)}$, and every circuit $C : \{0,1\}^n \times \{0,1\}^h \rightarrow \{0,1\}$ of size $\lambda^c$,

$$\Pr \left[ V(\tau, x, \pi) = 1 \mid (x, a) \notin R \right] \leq \frac{1}{2} \left[ \Pr \left[ (\sigma, \tau) \leftarrow G(1^\lambda, C) \mid (x, \pi) \leftarrow P^*(z, \sigma, \tau) \mid a \leftarrow E(z, \sigma, \tau) \right] \leq \text{negl}(\lambda) \right].$$

A zero-knowledge SNARK (or “succinct NIZK of knowledge”) is a SNARK satisfying a zero-knowledge property. Namely, zero knowledge ensures that the honest prover can generate valid proofs for true theorems without leaking any information about the theorem beyond the fact that the theorem is true (in particular, without leaking any information about the assignment that he used to generate the proof). Of course, when considering zero-knowledge SNARKs, the prover must trust the generation of the proving key $\sigma$. (Thus, combined with the fact that the verifier must trust the generation of the verification key $\tau$, the key generator $G$ must be run by a party that is trusted by both the prover and verifier.)

Definition 3.1.3. A triple of algorithms $(G, P, V)$ is a (perfect) zero-knowledge SNARK (for circuit satisfiability) if it is a SNARK (for circuit satisfiability) and, moreover, satisfies the following property:

Zero Knowledge

There exists a stateful interactive polynomial-size simulator $S$ such that for all stateful interactive polynomial-size distinguishers $D$, constant $c$, large enough security parameter $\lambda \in \mathbb{N}$, every auxiliary input $z \in \{0,1\}^{\text{poly}(\lambda)}$, and every circuit $C : \{0,1\}^n \times \{0,1\}^h \rightarrow \{0,1\}$ of size $\lambda^c$,

$$\Pr \left[ (x, a) \in R_C \mid D(\pi) = 1 \right] \leq \Pr \left[ (x, a) \leftarrow P(\sigma, x, a) \mid (x, a) \leftarrow D(\sigma, \pi) \mid \pi \leftarrow S(1^\lambda, C) \right].$$

As usual, Definition 3.1.3 can be relaxed to consider the case in which the distributions are only statistically or computationally close.

3.2 Definition of Linear PCPs

In Section 2.2 we informally introduced a linear PCP to be a PCP where the honest proof oracle is a linear function (over some underlying field), and soundness is required to hold only for linear proof oracles. For completeness, we give here formal definitions. For more details, see [BCIOP13].

A linear probabilistically-checkable proof (linear PCP) system for a relation $\mathcal{R}$ over a field $\mathbb{F}$ is one where the PCP oracle is restricted to compute a linear function $\pi : \mathbb{F}^m \rightarrow \mathbb{F}$ of the verifier’s queries. Viewed as a traditional PCP, $\pi$ has length $|\mathbb{F}|^m$ (and alphabet $\mathbb{F}$).
For simplicity, we ignore the computational complexity issues in the following definition, and refer to them when they are needed.

**Definition 3.2.1.** Let $\mathcal{R}$ be a binary relation, $F$ a finite field, $P_{LPCP}$ a deterministic prover algorithm, and $V_{LPCP}$ a probabilistic oracle verifier algorithm. We say that the pair $(P_{LPCP}, V_{LPCP})$ is a (input-oblivious) $k$-query linear PCP for $\mathcal{R}$ over $F$ with knowledge error $\varepsilon$ and query length $m$ if it satisfies the following requirements.

1. **Syntax.** On any input $x$ and oracle $\pi$, the verifier $V^\pi_{LPCP}(x)$ makes $k$ input-oblivious queries to $\pi$ and then decides whether to accept or reject. More precisely, $V_{LPCP}$ consists of a probabilistic query algorithm $Q_{LPCP}$ and a deterministic decision algorithm $D_{LPCP}$ working as follows. Based on its internal randomness, and independently of $x$, $Q_{LPCP}$ generates $k$ queries $q_1, \ldots, q_k \in F^m$ to $\pi$ and state information $u$; then, given $x$, $u$, and the $k$ oracle answers $a_1 = \langle \pi, q_1 \rangle, \ldots, a_k = \langle \pi, q_k \rangle$, $D_{LPCP}$ accepts or rejects.

2. **Completeness.** For every $(x, w) \in \mathcal{R}$, the output of $P_{LPCP}(x, w)$ is a description of a linear function $\pi : F^m \to F$ such that $V^\pi_{LPCP}(x)$ accepts with probability 1.

3. **Knowledge.** There exists a knowledge extractor $E_{LPCP}$ such that for every linear function $\pi^* : F^m \to F$ if the probability that $V^\pi^*_{LPCP}(x)$ accepts is greater than $\varepsilon$ then $E^\pi^*_{LPCP}(x)$ outputs $w$ such that $(x, w) \in \mathcal{R}$.\(^2\)

\(^2\)In particular, $(P_{LPCP}, V_{LPCP})$ has soundness error $\varepsilon$: for every $x$ such that $(x, w) \notin \mathcal{R}$ for all $w$, and for every linear function $\pi^* : F^m \to F$, the probability that $V^\pi^*_{LPCP}(x)$ accepts is at most $\varepsilon$.

An important efficiency measure for a linear PCP is the (algebraic) degree of the verifier. Specifically, we say that $(P_{LPCP}, V_{LPCP})$ has degree $(d_Q, d_D)$ if, additionally the following two requirements hold:

- The query algorithm $Q_{LPCP}$ has degree $d_Q$. Namely, there are $k$ polynomials $p_1, \ldots, p_k : F^n \to F^m$ and state polynomial $p : F^n \to F^{m'}$, all of degree $d_Q$, such that the linear PCP queries are $q_1 = p_1(r), \ldots, q_k = p_k(r)$ and the state is $u = p(r)$, for a random $r \in F^n$.

- The decision algorithm $D_{LPCP}$ has degree $d_D$. Namely, for every input $x$ there is a polynomial $t_x : F^{m'+k} \to F^n$ of degree $d_D$ such that $t_x(u, a_1, \ldots, a_k) = 0^n$ if and only if $D_{LPCP}(x, u, a_1, \ldots, a_k)$ accepts.

![Figure 3-3: Diagram of a $k$-query linear PCP of length $m$.](image)
In such a case, we call \( m' \), which is the number of field elements in the \( u \), the state length.

**Honest-verifier zero-knowledge linear PCPs.** We also consider honest-verifier zero-knowledge (HVZK) linear PCPs.

**Definition 3.2.2.** A PCP system \((P_{\text{PCP}}, V_{\text{PCP}})\) for a relation \( R \), where \( P_{\text{PCP}} \) is also probabilistic, is \( \delta \)-statistical HVZK if there exists a simulator \( S_{\text{PCP}} \), running in expected polynomial time, for which the following two ensembles are \( \delta \)-close (\( \delta \) can be a function of the field, input length, and so on):

\[
\{ S_{\text{PCP}}(x) \}_{(x,w) \in R} \text{ and } \{ \text{View}(V_{\text{PCP}}^\pi(x)) \mid \pi_{x,w} \leftarrow P_{\text{PCP}}(x, w) \}_{(x,w) \in R},
\]

where View represents the view of the verifier, including its coins and the induced answers according to \( \pi \).

### 3.3 Constructing a SNARKs from Linear PCPs

The transformation of Bitansky et al. [BCIOP13] 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 \( q \) consisting of \( m' = (k + 1)m \) elements in \( \mathbb{F} \); the message \( q \) 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 \( q \), i.e., reply with a vector \( a^* \in \mathbb{F}^{k+1} \) such that \( a^* = \Pi^*q + b^* \) for some \( \Pi^* \in \mathbb{F}^{(k+1) \times m'} \) and \( b^* \in \mathbb{F}^{k+1} \). The honest prover simply returns the vector \( a = (a_1, \ldots, a_{k+1}) \) where \( a_i = \langle \pi, q_i \rangle \), \( q_i \) is the \( i \)-th \( m \)-element block of \( q \), and \( \pi \) is the linear PCP proof. A prover’s message \( a^* \) is verified by checking consistency of \( a_{k+1}^* \) with \( a_1^*, \ldots, a_k^* \) and then invoking the linear PCP decision predicate on \( a_1^*, \ldots, a_k^* \); the consistency check ensures that \( a_i^* = \langle \pi^*, q_i \rangle \) for some linear PCP \( \pi^* \).

- **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 \( \text{Enc}(\cdot) \) with the following properties.

  1. It allows public testing of quadratic predicates on encoded elements.
  2. It provides a certain notion of one-way security to encoded elements.
3. It ensures that any polynomial-size prover can only perform linear operations on the encoded elements, “up to” information leaked by the encoding. Looking ahead, this leakage won’t affect the zero knowledge properties of our system, as our Linear PCP (before the encoding) is statistical HVZK and the queries of LPCP verifier are output by the trusted generator.

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

The encoding $\text{Enc}(\cdot)$ needed for Step 2 can be based on knowledge-of-exponent assumptions [Dam92, HT98, BP04], and requires us to fix $\mathbb{F} = \mathbb{F}_r$ for some prime $r$. Also, from the discussion above it is not clear why the elements in $q$ need to be random evaluations of low-degree polynomials; this requirement arises, for security reasons, in Step 2.

### 3.4 An Efficient Linear PCP

In this section, we discuss the problem of implementing a linear PCP for arithmetic circuit satisfiability that is as efficient as possible. First, let us recall that the circuit satisfaction problem of a circuit $C : \mathbb{F}^n \times \mathbb{F}^h \to \mathbb{F}^m$ is the relation $R_C = \{(x, a) \in \mathbb{F}^n \times \mathbb{F}^h : C(x, a) = 0^m\}$; its language is $L_C = \{x \in \mathbb{F}^n : \exists a \in \mathbb{F}^h, C(x, a) = 0^m\}$.

**Our linear PCP.** Our technical starting point for constructing a linear PCP for $R_C$ is the work on quadratic-span programs (QSPs) and quadratic-arithmetic programs (QAPs) of Gennaro et al. [GGPR13]. Indeed, Bitansky et al. [BCIOP13] 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 linear PCP for the relation $R_C$ that trades an increased number of 5 queries for a construction that, while keeping essentially-optimal asymptotics, enjoys excellent efficiency in practice.

Concretely, for checking membership in the language $L_C$ 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

---

3Since 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 about the underlying elements is necessarily leaked.
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 $2|C|/|F|$. Using suitable FFTs, the honest prover can compute the linear proof oracle via an arithmetic circuit of size $O(|C| \log |C|)$ and depth $O(\log |C|)$. (In particular, the prover is highly parallelizable.)

**Efficiency optimizations.** While there exists a variety of FFT algorithms, the most efficient ones are tailored to fields with special structure. With this in mind, we choose the prime $r$, which determines the field $F_r$ for the linear PCP, so that $r - 1 = 2^\ell m$ for a “large enough” integer $\ell$. Then, $F_r$ contains a primitive $2^\ell$-th root of unity, so multi-point evaluation/interpolation over domains consisting of roots of unity (or their multiplicative cosets) can be performed via a simple and efficient radix-2 FFT algorithm. This results in the aforementioned complexity for the honest prover. Furthermore, working over such $F_r$ simplifies the linear-time algorithm for sampling queries.

More precisely, when working with the language $L_C$, we need $2^\ell > |C|$ to hold. In practice, $\ell \geq 30$ seems adequate for the problem sizes of interest, so we chose $\ell = 30$ in our implementation. Larger values of $\ell$ can be substituted to support circuits $C$ with $|C| > 2^{30}$ (see Section 3.5).

**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.

Thus, we need to ensure that our linear PCP is HVZK. Bitansky et al. [BCIOP13] 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 is made HVZK with essentially no computational overhead, via a simple modification analogous to the one used in [GGPR13] to achieve zero knowledge. Thus, the SNARK obtained from our linear PCP has (statistical) zero knowledge.

We describe the HVZK linear PCP for circuit satisfiability that we designed and implemented. The basic design of our linear PCP builds on the quadratic arithmetic programs (QAPs) of Gennaro et al. [GGPR13].

Rather than directly constructing a linear PCP for circuit satisfiability, we first construct linear PCPs for a notationally more convenient language: satisfiability of systems of rank-1 quadratic equations over a finite field $F$. As discussed below, both Boolean and arithmetic circuit satisfiability are reducible to this language essentially without any overheads.

**Definition 3.4.1.** A system of rank-1 quadratic equations over $F$ is a tuple $S = ((a_j, b_j, c_j)_{j=1}^{N_g}, n)$ where $a_j, b_j, c_j \in F^{1+N_w}$ and $n \leq N_w$. Such a system $S$ is satisfiable with an input $x \in F^n$ if there is a witness $w = (w_1, \ldots, w_{N_w}) \in F^{N_w}$ such that:

1. $x = (w_1, \ldots, w_n)$, and
2. $\langle a_j, (1, w) \rangle \cdot \langle b_j, (1, w) \rangle = \langle c_j, (1, w) \rangle$ for all $j \in [N_g]$.

---

4 Here by FFT we mean transformation from the representation of a polynomial as a formal sum (or a sequence of coefficients), to a representation that is the evaluation of the polynomial on certain points in the field. However, the FFT algorithms used for finite fields are very similar to those invoked for time-based to frequency-based transforms (also referred to as FFTs).

5 While requiring that $r - 1$ be smooth complicates the search for an elliptic curve satisfying all of our requirements and the search’s complexity grows with $2^\ell$ (see Section 3.3), smoothness is crucial for leveraging tailored FFT algorithms in the prover.
In such a case, we write $S(x, w) = 1$. We call $N_g$ the number of constraints, $N_w$ the number of variables, and $n$ the input size.

The name rank-1 quadratic equation comes from quadratic forms $x^T M x$. In our case the corresponding matrices $M$ have rank 1, so we call the equations that force our rank-1 quadratic forms to take value 0 “rank-1 quadratic equations”.

**Definition 3.4.2.** The satisfaction problem of a system (of rank-1 quadratic equations) $S$ is the relation $R_S = \{(x, w) \in \mathbb{F}^n \times \mathbb{F}^{N_w} : S(x, w) = 1\}$; its language is denoted $L_S$.

A Boolean circuit $C: \{0, 1\}^n \times \{0, 1\}^h \rightarrow \{0, 1\}$ with $\alpha$ wires and $\beta$ gates induces a corresponding system of quadratic equations $S$ with $N_w = \alpha$ variables and $N_g = \beta + h + 1$ constraints. (The $h + 1$ additional constraints are to ensure that the $h$ wires corresponding to witness wires have boolean values $\{0, 1\}$ over (much larger!) field $F$, and that the output gate outputs 0.)

Similarly, an arithmetic circuit $C: \mathbb{F}^n \times \mathbb{F}^h \rightarrow \mathbb{F}^m$ with $\alpha$ wires and $\beta$ (bilinear)$^6$

Thus, we can focus on the relation $R_S$ without loss in generality or efficiency.

We prove the following claim:

**Claim 3.4.3.** For any finite field $\mathbb{F}$, there is a $5$-query linear PCP $(P_{LPCP}, (Q_{LPCP}, D_{LPCP}))$ for $R_S$ over $\mathbb{F}$ with knowledge error $\frac{2N_g}{|\mathbb{F}|}$, query length $5 + N_w + N_g$, state length $n + 2$, and degree $(d_Q, d_D) = (N_g, 2)$. Moreover, $Q_{LPCP}$ needs to sample only a single random element of $\mathbb{F}$ to generate the output queries and state. Finally, the linear PCP is $\frac{N_g}{|\mathbb{F}|}$-statistical HVZK.

**Remark 3.4.4.** As Bitansky et al. [BCIOP13] observed, the work of Gennaro et al. [GGPR13] implies various constructions of efficient linear PCPs. Specifically, any quadratic span program (QSP) for a relation $R$ yields a corresponding 3-query linear PCP for $R$, and any quadratic arithmetic program (QAP) for a relation $R$ yields a corresponding 4-query linear PCP for $R$.

Efficient constructions of both QSPs and QAPs have the same asymptotic efficiency, but in this work we build on the QAP approach. Indeed, while QAPs yield linear PCPs with 4 queries instead of 3 (a minor overhead), QAPs are significantly simpler to construct than QSPs, resulting in very small hidden constants. This property is crucial for practical applications. In fact, in our linear PCP construction we rely on an additional query (for a total of 5 queries) in order to further simplify our construction.

### 3.4.1 The Construction

We now describe the construction for $(P_{LPCP}, (Q_{LPCP}, D_{LPCP}))$ from Claim 3.4.3. (Later, in Section 3.4.2, we discuss how to implement these algorithms efficiently.) We begin by introducing some notation. Recall that we have fixed a system of quadratic equations $S = \{(a_j, b_j, c_j)_{j=1}^{N_g}, n\}$, where $a_j, b_j, c_j \in \mathbb{F}^{1+N_w}$ and $n \leq N_w$, and we are interested in the relation $R_S$.

---

$^6$A gate with inputs $x_1, \ldots, x_n \in \mathbb{F}$ is **bilinear** if the output is $\langle a, (1, x_1, \ldots, x_n) \rangle \cdot \langle b, (1, x_1, \ldots, x_n) \rangle$ for some $a, b \in \mathbb{F}^{n+1}$, where $\langle , \rangle$ denotes inner product. In particular, these include addition, multiplication, and constants gates.
Fix an arbitrary subset $S$ of $\mathbb{F}$ with $|S| = N_g$; let $S = \{\alpha_1, \ldots, \alpha_{N_g}\}$. For $i \in \{0, 1, \ldots, N_w\}$, define the three functions $A_i, B_i, C_i : S \to \mathbb{F}$ as follows: for each $j \in [N_g]$,

$$A_i(\alpha_j) := a_j(i), \quad B_i(\alpha_j) := b_j(i), \quad C_i(\alpha_j) := c_j(i),$$

where $a_j(i), b_j(i), c_j(i)$ denotes the $i$-th coefficients of vectors $a_j, b_j, c_j$ respectively.

Then extend each function $A_i, B_i, C_i$ into a degree-$(N_g - 1)$ univariate polynomial over $\mathbb{F}$, via interpolation. Also define $Z_S$ to be the unique $N_g$-degree univariate polynomial over $\mathbb{F}$ that vanishes on $S$.

The linear PCP prover $P_{\text{LPCP}}$ (when given suitable inputs) generates a vector of field elements $\pi$ that represents his choice of (honest) linear proof oracle; we now specify how $P_{\text{LPCP}}$ constructs $\pi$.

**Construction 3.4.5** (linear PCP prover algorithm). Given an input $x \in \mathbb{F}^n$ and a witness $w \in \mathbb{F}^{N_w}$ such that $(x, w) \in R_S$, the prover $P_{\text{LPCP}}$ works as follows:

1. draw $\delta_1, \delta_2, \delta_3$ independently at random from $\mathbb{F}$;

2. let $h = (h_0, h_1, \ldots, h_{N_g}) \in \mathbb{F}^{N_g + 1}$ be the coefficients of the univariate polynomial

$$H(z) := \frac{A(z)B(z) - C(z)}{Z_S(z)}, \quad (3.1)$$

of degree $N_g$, where $A, B, C$ are the univariate polynomials of degree $N_g$ that are defined as follows:

$$A(z) := A_0(z) + \sum_{i=1}^{N_w} w_i A_i(z) + \delta_1 Z_S(z),$$

$$B(z) := B_0(z) + \sum_{i=1}^{N_w} w_i B_i(z) + \delta_2 Z_S(z),$$

$$C(z) := C_0(z) + \sum_{i=1}^{N_w} w_i C_i(z) + \delta_3 Z_S(z);$$

3. output the vector $\pi \in \mathbb{F}^{3 + (N_w + 1) + (N_g + 1)}$ given by $(\delta_1, \delta_2, \delta_3, 1, w, h)$.

Note that $H(z)$ is indeed a polynomial: one can verify that $(x, w) \in R_S$ implies that $Z_S(z)$ divides $A(z)B(z) - C(z)$. Next, we describe the linear PCP query generator $Q_{\text{LPCP}}$.

**Construction 3.4.6** (linear PCP query algorithm). The query generator $Q_{\text{LPCP}}$ works as follows:

1. draw $\tau$ at random from $\mathbb{F}$;

2. output 5 queries $q_1, \ldots, q_5$ (of $5 + N_w + N_g$ field elements each), constructed as follows:
3. output the state \( \mathbf{u} = (u_1, \ldots, u_{n+2}) \) where \( u_i := \tau^{i-1} \) for \( i \in \{1, \ldots, n+1\} \) and \( u_{n+2} := Z_S(\tau) \).

Finally, the linear PCP decision algorithm \( D_{LPCP} \), given an input \( \mathbf{x} \), checks whether \( \mathbf{x} \in \mathcal{L}_S \), by relying on the state information \( \mathbf{u} \) produced by the query algorithm as well as the 5 field elements \( a_1 = \langle \pi^*, q_1 \rangle, \ldots, a_5 = \langle \pi^*, q_5 \rangle \), which are the answers when given a linear proof oracle \( \pi^* \) (potentially maliciously generated by a dishonest prover).

**Construction 3.4.7** (linear PCP decision algorithm). Given an input \( \mathbf{x} \in \mathbb{F}^n \), the state information \( \mathbf{u} = (u_1, \ldots, u_{n+2}) \), and answers \( (a_1, \ldots, a_5) \), the LPCP verifier \( D_{LPCP} \) accepts if and only if

\[
a_1a_2 - a_3 - a_4u_{n+2} = 0 \quad \text{and} \quad a_5 - u_1 - \sum_{i=1}^{n} x_iu_{i+1} = 0 .
\]

We only sketch the (simple) proof of Claim 3.4.3.

**Proof sketch of Claim 3.4.3.** First, it is clear that the linear PCP we just described has 5 queries of \( 5 + N_w + N_g \) elements each, and that the state information passed from \( Q_{LPCP} \) to \( D_{LPCP} \) contains \( n + 2 \) field elements. Regarding the degree of \( Q_{LPCP} \): each coordinate of any query generated by \( Q_{LPCP} \) is the evaluation of a polynomial on the (random) field element \( \tau \); each such polynomial (e.g., \( Z_S, A \), and so on) has degree at most \( N_g \). Regarding the degree of \( D_{LPCP} \), it is clear that \( D_{LPCP} \) tests the zero of two polynomials of degree 2. Overall, we deduce that the linear PCP has degree \( (d_Q, d_P) = (N_g, 2) \).

The knowledge error can be argued as follows: suppose that, for some \( \mathbf{x} \), a cheating prover produces a vector \( \pi^* = (\delta_1^*, \delta_2^*, \delta_3^*, \rho^*, \mathbf{w}^*, \mathbf{h}^*) \in \mathbb{F}^{3+(N_w+1)+(N_g+1)} \) such that

\[
\Pr_{\tau \leftarrow \mathbb{F}} \left[ V_{LPCP} \left( \mathbf{x}, \mathbf{u}, \langle \pi^*, q_1 \rangle, \ldots, \langle \pi^*, q_5 \rangle \right) = 1 \mid (q_1, \ldots, q_5, \mathbf{u}) \leftarrow Q_{LPCP}(\tau) \right] > \frac{2N_g}{|\mathbb{F}|} .
\]

By construction of \( Q_{LPCP} \) and \( D_{LPCP} \), we know the above equation is equivalent to:

\[
\Pr_{\tau \leftarrow \mathbb{F}} \begin{bmatrix}
A^*(\tau)B^*(\tau) - C^*(\tau) = Z_S(\tau) \cdot \left( \sum_{i=0}^{N_g} h_i^* \tau^{i-1} \right)
\end{bmatrix}
\quad \text{and} \quad \\
\rho^* + \sum_{i=1}^{n} w_i^* \tau^i = 1 + \sum_{i=1}^{n} x_i^i \tau^i
\]

\[
> \frac{2N_g}{|\mathbb{F}|} ,
\]

where

\[
A^*(z) := \rho^* A_0(z) + \sum_{i=1}^{N_w} w_i^* A_i(z) + \delta_1^* Z_S(z) ,
\]

\[
B^*(z) := \rho^* B_0(z) + \sum_{i=1}^{N_w} w_i^* B_i(z) + \delta_2^* Z_S(z) ,
\]

and

\[
C^*(z) := \rho^* C_0(z) + \sum_{i=1}^{N_w} w_i^* C_i(z) + \delta_3^* Z_S(z) .
\]
\[ B^*(z) := \rho^* B_0(z) + \sum_{i=1}^{N_w} w_i^* B_i(z) + \delta_2^* Z_S(z) , \]

\[ C^*(z) := \rho^* C_0(z) + \sum_{i=1}^{N_w} w_i^* C_i(z) + \delta_3^* Z_S(z) . \]

Thus, because the equalities involve polynomials of low-enough degree, we can deduce that they hold as polynomial identities (in the formal variable \( z \)). In particular, we deduce that \( \rho^* = 1 \), that \( \mathbf{x} = (w_1, \ldots, w_n) \). We similarly deduce that \( A^*(z) B^*(z) - C^*(z) \) vanishes everywhere on \( S = \{ \alpha_1, \ldots, \alpha_{N_g} \} \) and thus, by expanding terms, that \( (A_0(z) + \sum_{i=1}^{N_w} w_i^* A_i(z)) \cdot (B_0(z) + \sum_{i=1}^{N_w} w_i^* B_i(z)) - (C_0(z) + \sum_{i=1}^{N_w} w_i^* C_i(z)) \) also vanishes everywhere on \( S \). By construction of the polynomials \( A_i, B_i, \text{ and } C_i \), we conclude that \( \langle \mathbf{a}_j, (1, \mathbf{w}^*) \rangle \cdot \langle \mathbf{b}_j, (1, \mathbf{w}^*) \rangle = \langle \mathbf{c}_j, (1, \mathbf{w}^*) \rangle \) for each \( j \in [N_g] \), and thus that \( (\mathbf{x}, \mathbf{w}^*) \in \mathcal{R}_S \), as desired. (In particular, the output of the knowledge extractor \( E^*_{\mathcal{PCP}}(x) \) is defined to be \( \mathbf{w}^* \).)

Finally, \( \frac{N_g}{| \mathbb{F} |} \)-statistical HVZK for the honest prover \( P_{\mathcal{PCP}} \) can be argued as follows. If \( Z_S(\tau) \neq 0 \), because \( \delta_1, \delta_2, \delta_3 \) are selected uniformly and independently at random from \( \mathbb{F} \), it holds that \( a_1, a_2, a_3 \) are uniform and independent field elements in \( \mathbb{F} \) (and thus do not leak any information about \( \mathbf{w} \)); for a random \( \tau \), it holds that \( Z_S(\tau) \neq 0 \) with probability \( 1 - \frac{N_g}{| \mathbb{F} |} \). Also, \( a_4 \) is determined by \( a_1, a_2, a_3, u_{n+2} \) via the constraint \( a_1 a_2 - a_3 - a_4 u_{n+2} = 0 \), so that \( a_4 \) does not leak any additional information.

As for \( a_5 \), it only contains information about the part of \( w \) that is equal to \( x \), which is known to the verifier. Thus, overall, \( (a_1, \ldots, a_5, u) \) is a distribution that is \( \frac{N_g}{| \mathbb{F} |} \)-far from one that is independent of \( \mathbf{w} \).

### 3.4.2 Computational Complexity of the Construction

We discuss efficiency considerations for the linear PCP \( (P_{\mathcal{PCP}}, (Q_{\mathcal{PCP}}, D_{\mathcal{PCP}})) \) for \( \mathcal{R}_S \) that we just presented. The decision algorithm \( D_{\mathcal{PCP}} \) is only testing two simple quadratic equations, and computing its decision bit is already very efficient: it only involves \( 2n + 9 \) field operations. Therefore, our discussion below focuses on minimizing the complexity of computing the query algorithm \( Q_{\mathcal{PCP}} \) and the prover algorithm \( P_{\mathcal{PCP}} \).

We begin by ensuring that we work in a field \( \mathbb{F} \) with a nice algebraic structure. Specifically, we assume that \( N_g \) is a power of 2 (this can be achieved by adding dummy constraints) and that \( \mathbb{F} \) has a \( N_g \)-th root of unity. So, in the sequel, we fix \( \omega \) to be a principal \( N_g \)-th root of unity, and we choose \( S = \{ \alpha_1, \ldots, \alpha_{N_g} \} \) with \( \alpha_i = \omega^{i-1} \).

**Computing the query algorithm.** The complexity of computing \( Q_{\mathcal{PCP}} \) is dominated by the complexity of evaluating each \( A_i, B_i, C_i \) at the (random) element \( \tau \).

We first explain how to efficiently compute \( A_0, \ldots, A_{N_w} \); a similar discussion holds for the \( B_i \) and \( C_i \). Recall the formula for Lagrange interpolation:

\[ A_i(z) := \sum_{j=1}^{N_g} \mathbf{a}_j(i) \cdot L_j(z) , \text{ where } L_j(z) := \frac{\prod_{k \neq j} (z - \alpha_k)}{\prod_{k \neq j} (\alpha_j - \alpha_k)} . \]
We can also write:

\[ A_i(z) = \sum_{j=1}^{N_k} a_j(i) \cdot \frac{L_j'(z)}{z - \alpha_j}, \text{ where } L_j'(z) := \frac{Z_S(z)}{\prod_{k \neq j} (\alpha_j - \alpha_k)}. \]

Because \( S = \{\alpha_1, \ldots, \alpha_{N_g}\} = \{1, \omega, \ldots, \omega^{N_k-1}\} \) are the \( N_g \)-th roots of unity, computing the \( L_j' \) is particularly easy. First, \( Z_S(z) = z^{N_k} - 1 \). Moreover, when \( \alpha_k \) ranges over all roots of unity that are not equal to \( \alpha_j \), the expression \( \omega \alpha_k \) ranges over all roots of unity that are not equal to \( \omega \alpha_j = \alpha_{j+1} \). Therefore,

\[
\frac{Z_S(z)}{L_j'(z)} = \prod_{k \neq j+1} (\alpha_j + 1 - \alpha_k) = \omega^{N_k-1} \cdot \prod_{k \neq j} (\alpha_j - \alpha_k) = \frac{1}{\omega} \cdot \frac{Z_S(z)}{L_j'(z)},
\]

and we deduce that \( L_{j+1}'(z) = \omega \cdot L_j'(z) \). Thus, if we compute \( L_1'(\tau) \), then we can compute \( L_2'(\tau), \ldots, L_{N_g}'(\tau) \) with only \( N_g - 1 \) additional multiplications.

We claim that \( L_1'(z) = Z_S(z)/N_g \); this can be seen as follows. The polynomial \( z^{N_k} - 1 \) can be (always) factored as \((z - 1) \cdot (1 + z + \cdots + z^{N_k-1}) \) or (in the field \( \mathbb{F} \) used here) as \((z - 1) \cdot (z - \omega) \cdot \cdots (z - \omega^{N_k-1}) \). We deduce that \( 1 + z + \cdots + z^{N_k-1} = (z - \omega) \cdot \cdots (z - \omega^{N_k-1}) \).

By setting \( z = 1 \), we conclude that \( N_g = (1 - \omega) \cdot \cdots (1 - \omega^{N_k-1}) = Z_S(z)/L_1'(z) \), as claimed.

Overall, we obtain an algorithm that, given \( \tau \) (as well as \( \omega \) and \( N_g \)), outputs \( L_j(\tau) = \frac{L_j'(\tau)}{\tau - \alpha_j} \) for \( j = 1, \ldots, N_g \) by using only \( 4N_g + \log N_g \) field operations. Specifically, the algorithm is as follows:

1. \( \zeta \leftarrow \tau^{N_k} - 1; \)
2. \( \lambda \leftarrow \zeta/N_g; \)
3. \( \rho \leftarrow 1 \)
4. \( L_1(\tau) \leftarrow \lambda/(\tau - \rho) \)
5. for \( j \in \{2, \ldots, N_g\} \):
   (a) \( \lambda \leftarrow \omega \lambda; \)
   (b) \( \rho \leftarrow \omega \rho; \)
   (c) \( L_j(\tau) \leftarrow \lambda/(\tau - \rho); \)
6. output \( L_1(\tau), \ldots, L_{N_g}(\tau). \)

Then, after computing \( L_1(\tau), \ldots, L_{N_g}(\tau) \), computing \( A_0(\tau), \ldots, A_{N_0}(\tau) \) only requires taking appropriate linear combinations of these, as determined by the coefficient vectors \( a_1, \ldots, a_{N_g} \). Specifically, the number of field operations to compute all the necessary linear combinations is \( 2 \sum_{j=1}^{N_g} ||a_j||₀ \), where \( ||a_j||₀ \) denotes the number of non-zero coordinates in the vector \( a_j. \)
Recalling the definition of \( Q_{\text{LPCP}} \) (which involves evaluating each \( A_i, B_i, C_i \) at a point \( \tau \), and a few other small computations on \( \tau \)), one can see that computing the outputs of \( Q_{\text{LPCP}} \) requires only
\[
4N_g + \log N_g + 2 \sum_{j=1}^{N_g} (||a_j||_0 + ||b_j||_0 + ||c_j||_0)
\]
field operations. When the quadratic system \( S \) is obtained, e.g., from a circuit \( C \) of fan-in 2, it holds that \( ||a_j||_0, ||b_j||_0, ||c_j||_0 = O(1) \) for each \( j \) so that computing \( Q_{\text{LPCP}} \) requires only \( O(|C|) \) field operations.

**Computing the prover algorithm.** The complexity of computing \( P_{\text{LPCP}} \) is dominated by the complexity of computing the coefficients of the \( N_g \)-degree polynomial \( H \) (see Equation 3.1). A natural approach to efficiently compute the coefficients of \( H \) is via a suitable use of FFTs over finite fields. We show how to do so “generically”, and then how to choose parameters so that we can leverage particularly simple and fast FFTs.

So let us begin by introducing notation for multipoint evaluation and interpolation. Given a domain \( D \subseteq \mathbb{F} \) and a polynomial \( A(z) \) of degree less than \( |D| \), we use \( \text{FFT}_D(A(z)) \) to denote a “generic” FFT that outputs the vector \((A(\alpha))_{\alpha \in D}\). Similarly, we use \( \text{FFT}_D^{-1}(A(\alpha))_{\alpha \in D} \) to denote the inverse operation (i.e., given \( |D| \) points, return the polynomial of degree less than \( |D| \) that interpolates between these points on \( D \)).

We now describe how to compute \( P_{\text{LPCP}} \) in terms of the above notation. Below, we let \( T \) be a subset of \( \mathbb{F} \) with \( |T| = N_g \) and \( S \cap T = \emptyset \); let \( T = \{\beta_1, \ldots, \beta_{N_g}\} \). Later we fix a convenient choice of \( T \). The algorithm of \( P_{\text{LPCP}} \) is as follows:

1. For \( j \in \{1, \ldots, N_g\} \), compute:
   \[
   A'(\alpha_j) := A_0(\alpha_j) + \sum_{i=1}^{N_w} w_i A_i(\alpha_j) = a_j(0) + \sum_{i=1}^{N_w} w_i a_j(i),
   \]
   \[
   B'(\alpha_j) := B_0(\alpha_j) + \sum_{i=1}^{N_w} w_i B_i(\alpha_j) = b_j(0) + \sum_{i=1}^{N_w} w_i b_j(i), \quad \text{and}
   \]
   \[
   C'(\alpha_j) := C_0(\alpha_j) + \sum_{i=1}^{N_w} w_i C_i(\alpha_j) = c_j(0) + \sum_{i=1}^{N_w} w_i c_j(i).
   \]

2. Compute the \((N_g - 1)\) coefficients of \( A'(z) \) by invoking \( \text{FFT}_S^{-1}(A'(\alpha_1), \ldots, A'(\alpha_{N_g})) \).
   Compute the \((N_g - 1)\) coefficients of \( B'(z) \) by invoking \( \text{FFT}_S^{-1}(B'(\alpha_1), \ldots, C'(\alpha_{N_g})) \).
   Compute the \((N_g - 1)\) coefficients of \( C'(z) \) by invoking \( \text{FFT}_S^{-1}(C'(\alpha_1), \ldots, C'(\alpha_{N_g})) \).

3. Compute the evaluation of \( A'(z) \) on \( T \) by invoking \( \text{FFT}_T(A'(z)) \).
   Compute the evaluation of \( B'(z) \) on \( T \) by invoking \( \text{FFT}_T(B'(z)) \).
   Compute the evaluation of \( C'(z) \) on \( T \) by invoking \( \text{FFT}_T(C'(z)) \).

4. Compute the evaluation of \( H'(z) := (A'(z)B'(z) - C'(z))/Z_S(z) \) on \( T \), point-by-point by using the evaluations of \( A'(z), B'(z), C'(z), Z_S(z) \) on \( T \).

5. Compute the \((N_g - 2)\) coefficients of \( H'(z) \) by invoking \( \text{FFT}_T^{-1}(H'(\beta_1), \ldots, H'(\beta_{N_g})) \).
6. Compute the \( N_g \) coefficients of \( H(z) := H'(z) + \delta_2 A'(\alpha) + \delta_1 B'(\alpha) + \delta_1 \delta_2 Z_S(z) - \delta_3 \), by directly evaluating the sum.

7. Output the \( N_g \) coefficients of \( H(z) \).

Step 1 can be performed with \( 2 \sum_{j=1}^{N_g} (||a_j||_0 + ||b_j||_0 + ||c_j||_0) \) field operations; Step 2, Step 3, and Step 5 all involve computing FFTs on a domain of size \( N_g \), and we will discuss their efficiency shortly; Step 4 involves computing \( Z_S(z) \) everywhere on \( T \), whose complexity we also discuss shortly, and then performing \( 4N_g \) field operations; Step 6 requires \( O(N_g) \) because all the requisite coefficients have already been computed.

We choose \( T \) to be a multiplicative coset of \( S \): for some \( \xi \in (\mathbb{F} \setminus S) \), we choose \( T := \xi S \). This choice greatly simplifies Step 2, Step 3, and Step 4, as follows.

First, \( Z_S(z) \) is equal to \( \xi^{N_g - 1} \) everywhere on \( T \). Therefore, evaluating \( Z_S(z) \) on \( T \) in Step 4 only requires \( 1 + \log N_g \) field operations.

Moreover, \( FFT_S^{-1} \) (for Step 2), \( FFT_T \) (for Step 3), and \( FFT_T^{-1} \) (for Step 5) are all FFTs (or inverse FFTs) that take only require \( O(N_g \log N_g) \) field operations, and have particularly nice algorithm for computing them. Specifically, letting \( \Xi \) be the diagonal matrix whose \( i \)-th diagonal entry is \( \xi^{i-1} \) and letting \( S^{-1} = \{1, \omega^{-1}, \ldots, \omega^{-N_g+1}\} \), it holds that:

\[
FFT_S^{-1}(\cdot) = FFT_{S^{-1}}(\cdot) \quad FFT_T(\cdot) = (FFT_S \circ \Xi)(\cdot) \quad FFT_T^{-1}(\cdot) = (\Xi^{-1} \circ FFT_S^{-1})(\cdot)
\]

As for \( FFT_S \), it is the “standard” FFT algorithm for finite fields that relies on an \( N_g \)-th root of unity (where \( N_g \) is a power of 2), and the main idea is to separately recurse on the even-power and odd-power coefficients of the polynomial and then suitably combine the answers.

In sum, \( P_{LPCP} \) can be computed with

\[
2 \sum_{j=1}^{N_g} (||a_j||_0 + ||b_j||_0 + ||c_j||_0) + O(N_g \log N_g)
\]

field operations. As before, when the quadratic system \( S \) is obtained, e.g., from a circuit \( C \) of fan-in 2, it holds that \( ||a_j||_0, ||b_j||_0, ||c_j||_0 = O(1) \) for each \( j \) so that computing \( P_{LPCP} \) requires only \( O(N_g \log N_g) \) field operations.

### 3.5 Optimizing the transformation from Linear PCPs to SNARKs

In the previous section we discussed how to implement a linear PCP that would be as efficient as possible. In this section we will describe how to ensure that the transformation from a linear PCP to a corresponding SNARK adds as little computational overhead as possible.

**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:

- The SNARK generator $G$, after sampling $q \in \mathbb{F}^{m'}$, must compute $\text{Enc}(q) = (\text{Enc}(q_i))_{i=1}^{m'}$. In other words $G$ has to compute the encoding of $m' = (k+1)m$ field elements.
- The honest SNARK prover $P$ must compute $\text{Enc}((\pi, q_i))$ for $i = 1, \ldots, k+1$, starting from $\text{Enc}(q)$ and the linear PCP proof $\pi \in \mathbb{F}^m$. In other words, $P$ has to homomorphically evaluate $k+1$ inner products.

In our case, the linear PCP we use (see Section 3.4) is over the field $\mathbb{F} = \mathbb{F}_r$, where $r$ is a 181-bit prime; the linear PCP has $k = 5$ queries and $m = \Theta(|C|)$ field elements per query. Furthermore, the encoding we use is $\text{Enc}(\gamma) = (g^\gamma, h^\gamma)$ where $g, h$ are, respectively, generators of groups $G_1, G_2$ of order $r$. The linear homomorphism is $\text{Enc}(a\gamma + b\delta) = \text{Enc}(\gamma)^a\text{Enc}(\delta)^b$ with coordinate-wise multiplication and exponentiation.

Therefore, $G$ and $P$ must compute a large number of exponentiations in $G_1, G_2$. These dominate the complexity of $G$ and $P$, and thus their efficiency is essential.

Efficiency optimizations. We address the aforementioned cryptographic bottlenecks as follows.

1. **Reducing the number of group operations in $P$.** The SNARK prover $P$ faces several large instances of a multi-exponentiation problem, a well-studied computational problem in applied cryptography [Ber02]. The problem is as follows: given group elements $g_1, \ldots, g_m \in G$ (here, $G = G_1$ or $G = G_2$) and integers $a_1, \ldots, a_m$, compute $\prod_{i=1}^m g_i^{a_i}$ (for our choice of parameters all $a_i$ are 181 bits long). In order to reduce the number of group operations required to compute this product, we implemented a suitable choice of multi-exponentiation algorithm [BC89]. Compared to the naive approach of “exponentiate and then multiply”, we save a multiplicative factor of 25 already for $m = 10^6$ (and the savings increase with $m$).

2. **Reducing the number of group operations in $G$.** The SNARK generator $G$ is instead faced with several large instances of the following exponentiation problem: given a group element $g \in G$ and $a_1, \ldots, a_m$, compute the tuple $(g^{a_1}, \ldots, g^{a_m})$ (for our choice of algebraic parameters all $a_i$ are 181 bits long, see Section 6.1). We reduce the number of required group operations by using the standard technique of pre-computing a table of powers of $g$, and then reusing these values in each subsequent exponentiation. We thus save a multiplicative factor of 23 in the number of group operations (over the naive approach of performing a “fresh” exponentiation for each term). Precomputing more powers of $g$ provides even greater savings, at the expense of more space usage.

3. **Reducing the cost of group operations.** We sought instantiations of the groups $G_1$ and $G_2$ that offer particularly efficient group operations.

A crucial requirement is that $G_1$ and $G_2$ must admit an efficient pairing (non-degenerate bilinear map) $e: G_1 \times G_2 \to \mathbb{G}_T$, where $\mathbb{G}_T$ is a “target” group (also of order $r$); indeed, the pairing $e$ provides the necessary functionality to publicly test quadratic predicates...
on encoded elements. We focus on asymmetric pairings (where $G_1 \neq G_2$), because of the extra flexibility in group choices at a given security level. Concretely, we work with the (reduced) Tate pairing [FR94, FMR06].

Thus, we need to find a suitable pairing-friendly elliptic curve $E$, defined over $\mathbb{F}_q$ for a prime $q$, and set $G_1, G_2, G_T$ equal to suitable subgroups of $E(\mathbb{F}_q), E(\mathbb{F}_{q^k}), \mathbb{F}_{q^k}^*$, respectively. (For a field extension $K$ of $\mathbb{F}_q$, $E(K)$ is the group of $K$-rational points on the curve $E$; and $k$ is known as $E$’s embedding degree.)

Concretely, in order to optimize the efficiency of multiplication and squaring in $G_1$ and $G_2$, we need to:

(a) minimize the number of operations in $\mathbb{F}_q$ needed for carrying out group multiplication and squaring; and

(b) do so without making $q$ much larger than $r$, that is, without making the value $\rho := \frac{\log q}{\log r}$ too large.

Furthermore, to allow for an efficient implementation of the underlying linear PCP (via suitable FFTs in $\mathbb{F}_r$; see Section 3.4), we require smoothness: $r - 1$ should be divisible by a “large enough” power of 2.

While finding an elliptic curve satisfying any of these requirements is not hard, finding an elliptic curve that simultaneously satisfies all the requirements is not easy because one can (heuristically) show that such elliptic curves are “rare”.

Our strategy for finding a suitable curve $E$ is outlined in the extended version of our paper [BCGTV13a]. In short, we consider a parametrized family of curves specified by a quadratic polynomial $q(x)$ and a linear polynomial $t(x)$ such that, for any integer $a$ such that $q(a)$ is prime and $|t(a)| \leq 2q(a)$, there is an elliptic curve $E$ over $\mathbb{F}_q$ with order $n(a) = q(a)t(a) + 1$ and embedding degree $k = 6$ (which makes the curve pairing friendly); in particular we consider family from [GMV07]. If the square-free part of $|4q(a)t(a)^2|$ is not large, E can be constructed via the method of complex multiplication [? ]. To address the smoothness requirement (i.e., the condition that $r1 = 2^m$) we iterate over fundamental discriminants in order of increasing magnitude and, for each fundamental discriminant $D$, we try to derive good solutions to $4q(x)t(x)^2 = y^2D$ by using a generalized Pell-equation solver. The authors are grateful to Andrew Sutherland for his generous guidance here.

4. Reducing the number of $G_2$ group operations in $G$ and $P$. Working with an asymmetric pairing causes $G_2$ operations to be about three times more expensive than $G_1$ operations. We modify the cryptographic transformation of [BCIO13], for the specific case of our linear IP construction, so that only a $\approx \frac{1}{10}$ fraction of the generator’s and prover’s group operations have to be carried out in $G_2$, while for the rest it suffices to carry them in $G_1$.

We conclude the discussion about efficiency optimizations by noting that the cryptographic computations in both the key generator and prover are highly-parallelizable; thus, their
latency can be greatly improved. Our prototype implementation does not seek to reduce latency or exploit parallelism. (See Section 6.4.)
Chapter 4

A New Quasi-linear Circuit Generator for RAM Machines

As summarized in Section 1.3, we implemented an efficient transformation that reduces correctness of program execution to circuit satisfiability. The following gives further design and performance details about this transformation. Concretely, in Section 4.1 we motivate and discuss our choice of architecture, TinyRAM. Then, in Section 4.2, we discuss implementation and performance of our compiler from C to TinyRAM assembly. Finally, in Section 4.3, we discuss implementation and performance of our reduction from the correctness of TinyRAM assembly to circuit satisfiability.

4.1 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 execution 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 and 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 instruction’s execution; 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). Yet, even these “simple” ISAs are quite rich: they support many addressing modes, many conditional branches, floating point arithmetic, instructions for parallel execution, and so on. For example, the ARM architecture has more than 35 SIMD instructions for addition and subtraction alone [ARM12]; also, even 8-bit versions of the Atmel AVR family support as much as 25 different conditional branch instructions alone [ATM10]. Once again, while these features reduce code size and execution time somewhat, they greatly increase circuit complexity.

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 $K$. (When we wish to make this explicit, we write $\text{TinyRAM}_{W,K}$. ) The global state of the machine at any time consists of:

- the *program counter*, denoted $\text{pc}$; it consists of $W$ bits;
- $K$ general-purpose *registers*, denoted $r_0, r_1, \ldots, r(K - 1)$, each consisting of $W$ bits;

¹This does not include the (crucial) task of checking the correctness of values loaded from random-access memory. Memory consistency is efficiently handled separately; see Section 4.3.
• 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 4.1.1 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 4.2, and, furthermore, allow for small circuits implementing its transition function (and other checks), as discussed in Section 4.3. 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 fairly simple load and store operations is important for efficiently verifying consistency of random-access memory; see Section 4.3.

In keeping with the setting of verifying 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.

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

**Definition 4.1.1.** Fix the word size \( W \) and number of registers \( K \). Let \( P \) be a TinyRAM\(_{W,K} \) program, let \( x \) and \( w \) be strings of \( W \)-bit words. We say that \( P(x, w) \) accepts in \( T \) steps if \( P \), with \( x \) as primary input and \( w \) as auxiliary input, executes the instruction accept in step \( T \).

The TinyRAM universal language is \( L_U = \bigcup_{W,K} L_{W,K} \), where \( L_{W,K} \) consists of the triples \( (P, x, T) \) where \( P \) is a TinyRAM\(_{W,K} \) 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 \( P(x, w) \) accepts in \( T \) steps.

A specification for the TinyRAM architecture can be found in [BCGTV13b].

### 4.2 A Compiler from C to TinyRAM

The GCC compiler [StGDC] 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.
void sumarray(int size,
    int* A,
    int* B,
    int* C)
{
    int i;
    for (i=0; i<size; i++) {
        C[i] = A[i] + B[i];
    }
}

Figure 4-1: Illustrative example of transforming a C program into TinyRAM assembly language.

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$^2$ of C to TinyRAM, with word size $W \in \{8, 16\}$ and number of registers $K \geq 16$. See Figure 4-1 for an illustrative example.

Because TinyRAM’s instruction set is quite minimal, any operation not directly supported by TinyRAM “hardware” (i.e., by a TinyRAM instruction) 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 instructions required to execute a piece of code). Initial experiments indicate 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 examples$^3$ into TinyRAM assembly using our GCC port, compared to the code produced by standard GCC for some common architectures: x86, ARM and AVR. We choose number of instructions (instead of, say, number of bytes) to be our metric as the size of the generated circuit is directly related to number of instructions, and therefore this is a more meaningful metric in verification setting. However, given upper and lower bounds of instruction sizes, similar results hold for number of bytes in the generated program. Figure 4-2 presents the results of compiling these examples.

The results show that, compared to the RISC architectures (ARM and AVR), the resulting TinyRAM code is at most three times larger than ARM and significantly smaller than AVR. Compared to x86, which is a very rich CISC architecture, TinyRAM code is up to four times bigger. We deduce that, at least for the program styles represented by these examples, the TinyRAM architecture allows for compilation into compact programs.

---

$^2$ Floating-point arithmetic and static data are not yet implemented, but pose no fundamental difficulty. Most of the C standard library is not yet implemented; see discussion below.

$^3$ The examples, described in Appendix A, are simple, natural C functions we wrote to demonstrate various program styles. They exercise memory accesses (pointer chasing and the RC4 stream cipher), integer arithmetic (matrix multiplication and polynomial evaluation), and logical calculations (single-source shortest paths and Game of Life).
Figure 4-2: Ratio of the number of instructions in the compiled program in TinyRAM to the number of instructions in the compiled program in other architectures (x86, ARM, AVR).
Figure 4-3: Ratio of the number of executed instructions in TinyRAM to the number of executed instructions in x86.
Execution time overhead. The circuits ultimately produced by our reduction have \(O(T \log T)\) gates, 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.

In the examples depicted in Figure 4-3, we observe that, in terms of execution time measured in number of executed instructions, 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, heavily optimized for minimizing instruction count, which is typically implemented using many millions of gates. (Recall the difference of executing vs. verifying, discussed in Section 4.1.)

These small overheads are more than compensated by the fact that TinyRAM has a very compact circuit that verifies the correctness of the transition function. For instance, for a word size \(W = 16\) and number of registers \(K = 16\), and for a program with 100 instructions, we obtain a 785-gate circuit for verifying the transition function.

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 number of instructions executed. In Section 4.3, we discuss the circuit complexity of TinyRAM’s transition function and how to efficiently verify TinyRAM execution traces.

Looking ahead. The C specification provides a standard library, which C programs liberally rely on. Our TinyRAM compiler at the moment does not provide support for this library and we are working on extending its functionality to include it, starting with functionality that does not interact with external environment, for example, string handling.

The two main challenges in providing full implementation are those functions that must be written directly in the underlying machine language, and supporting (or reasonably approximating) functionality that extends into the program runtime environment, such as file I/O, process management, inter-process communication (IPC), and other system services.

4.3 An Efficient Reduction from TinyRAM to Circuit Satisfiability

The following describes our efficient reduction from the correctness of TinyRAM executions to \(F\)-arithmetic circuit satisfiability, for any prime field \(F\) of sufficiently large size.

4.3.1 The reduction notion

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

**Definition 4.3.1.** 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 $(\text{circ}, \text{wit}, \text{wit}^{-1})$ such that, for every TinyRAM program $P$, time bound $T$, and primary input size $n$, the following hold:

- $C := \text{circ}(P, T, n)$ is an $\mathbb{F}$-arithmetic circuit from $\mathbb{F}^{W \cdot n} \times \mathbb{F}^h$ to $\mathbb{F}^m$ for some $h, m$; $C$’s gates are bilinear;
- for every $(x, w)$ such that $P(x, w)$ accepts in $T$ steps, $C(x, \text{wit}(P, T, x, w)) = 0^m$;
- for every $(x, a)$ such that $C(x, a) = 0^n$, $P(x, \text{wit}^{-1}(P, T, x, a))$ accepts in $T$ steps.

The work on fast reductions of Ben-Sasson et al. [BCGT13a] implies a reduction $(\text{circ}, \text{wit}, \text{wit}^{-1})$ where $|C|$ (the number of gates in $C$) is $O(T(\log T)^2)$ and $\text{circ}, \text{wit}, \text{wit}^{-1}$ 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 focus our attention only on the efficiency of the circuit and witness maps (i.e., $\text{circ}$ and $\text{wit}$), because these need to be run in practice.\(^5\) Before discussing our work, however, we briefly review the approach of [BCGT13a].

### 4.3.2 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 time step.

- The transition *function* of TinyRAM, denoted $\Pi_{\text{TF}}$, is the predicate that, given a TinyRAM program $P$ 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 $P$.\(^6\)

- An *execution trace*\(^7\) for a TinyRAM program $P$, time bound $T$, and primary input $x$ is a sequence of states $\text{tr} = (S_1, \ldots, S_T)$. An execution trace $\text{tr}$ is *valid* if there exists an auxiliary input $w$ such that the sequence of states induced by $P$ running with input tapes $(x, w)$ is $\text{tr}$.

---

\(^4\) Given a space bound $S$ on the computation of $P$ 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.

\(^5\) Concretely, the key generator runs $\text{circ}$ while the prover runs $\text{wit}$; see Figure 6-5 and Figure 6-6. In contrast, $\text{wit}^{-1}$ ensures that “proof of knowledge is preserved” (i.e., $\text{wit}^{-1}$ only appears as part of a proof of security). We are thus not particularly interested in optimizing $\text{wit}^{-1}$, especially because (just as in [BCGT13a]) it can be computed in time that is only $O(T(\log T)^2)$.

\(^6\) 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 (cf. discussion of executing vs. verifying in Section 4.1).

\(^7\) An execution trace is also at times known as a computation transcript [BCGT13a].
The goal is to design an \( F \)-arithmetic circuit \( C \) for verifying that \( \text{tr} \) 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 \( \Pi_{TF} \) of TinyRAM: namely, \( C_{TF}(P, S, S') = 1 \) if and only if \( \Pi_{TF}(P, S, S') = 1 \). By invoking \( C_{TF} \) on each pair of successive states of \( \text{tr} \), we can verify every state transition in the trace \( \text{tr} \), i.e., ensure that \( \Pi_{TF}(P_i, S_i, S_{i+1}) = 1 \) 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 \( \text{tr} \), checks that \( \text{tr} \) is code-consistent.

**Step 2: memory consistency.** The global state of a random-access machine also includes memory. In particular, in order to verify that \( \text{tr} \) is valid, we also need to verify that \( \text{tr} \) 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 \( P \) 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 snapshot of the entire machine state (which includes registers and memory) for each time step is not efficient: such a circuit has size \( \Omega(T^2) \). (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] take a more efficient approach, building 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 \( \text{tr} \) but sorted according to accessed memory addresses (and breaking ties via timestamps); let us denote this sorted trace by \( \text{MemSort}(\text{tr}) \). Concretely, one can define another “local” predicate \( \Pi_{MC} \) such that, if \( \Pi_{MC} \) is satisfied by each pair of adjacent states in \( \text{MemSort}(\text{tr}) \) (and, in addition, \( \text{tr} \) is code-consistent) then \( \text{tr} \) is valid. We can then augment \( C \) with \( T \) copies of a sub-circuit \( C_{MC} \) that verifies the predicate \( \Pi_{MC} \) on \( \text{MemSort}(\text{tr}) \). The circuit \( C \) is thus left to verify that the auxiliary input \( \text{MemSort}(\text{tr}) \) is the result of sorting \( \text{tr} \).

**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 \( \text{tr} \) into \( \text{MemSort}(\text{tr}) \). A \( T \)-packet routing network is a directed graph with \( T \) sources, \( T \) sinks, and inner nodes (switches) such that, for any permutation \( \pi: [T] \to [T] \), there are routing decisions for the switches that cause \( T \) packets at the sources to travel to the \( T \) sinks, according to the permutation \( \pi \), 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 \( T \cdot (|C_{TF}| + |C_{MC}| + O((\log T)^2)) \) for verifying the validity of a \( T \)-step trace.
Chapter 5
Design of algebraic time and memory consistency checkers

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 4.1) we devised TinyRAM to facilitate the design of a small circuit $C_{TF}$ for the transition function $\Pi_{TF}$, 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 $|P|$?

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 of large characteristic,\(^1\) and
2. nondeterministic advice

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 $\text{circ}$.

To illustrate the use of (1) and (2), consider the basic task of multiplexing bit vectors, used numerous times in $C$. Given $n$ vectors $a_1, \ldots, a_n$ of $\ell$ bits each, and a $\lceil \log n \rceil$-bit index $i$, we seek a small $\mathbb{F}$-arithmetic circuit that computes the vector selected by the index. A naive multiplexer circuit requires $\Theta(n(\ell + \log n))$ bilinear gates.\(^2\) In contrast, by relying on (1) and (2), we design a multiplexer circuit that needs only $O(n\ceil{\log \mathbb{F}})$ bilinear gates. The efficiency improvement is significant because we ultimately need to work with cryptographically-large fields; for instance, in our setting where $\mathbb{F} = \mathbb{F}_r$ and $r$ is an 181-bit prime, 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 $a_i$, as well as the index $i$, 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

\(^1\)As required by the underlying zk-SNARK for circuit satisfiability; see Section 3.5.

\(^2\)For example: for each $i = 1, \ldots, n$, multiply $a_i$ by 1 if $i$ represents $i$, and by 0 otherwise, using $\Theta(\log n + \ell)$ gates per $i$; then for $j = 1, \ldots, \ell$, produce the $j$-th output bit using one fan-in-$n$ adder per $j$. 
case, we could easily construct a nondeterministic \(Z\)-arithmetic circuit of size \(O(n)\) (with bilinear gates of unbounded fan-in): guess variables \(b_1, \ldots, b_n\) such that \(\sum_{i=1}^{n} b_i = 1\) and \(\{b_i \cdot (i-i) = 0\}_{i=1}^{n}\), and then output the inner product of the vector \((b_1, \ldots, b_n)\) and the vector \((a_1, \ldots, a_n)\). However, the \(a_i\) and \(i\) are only given to us as strings of bits, and we need to work with \(F\)-arithmetic circuits. This gap motivates two fundamental operations: packing and unpacking of bit vectors. Packing denotes mapping a bit vector (using one field element per bit) into a shorter sequence of field elements that represent those bits using a denser encoding; unpacking denotes the inverse operation. The packing operation is very efficient: in the prime field \(F_r\) with \(r \geq 2^\ell\), a single bilinear 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, \(r \geq 2^\ell\) need not hold, so we use \(\lceil \ell/\log_2 r \rceil\) field elements to store an \(\ell\)-bit vector. Given the aforementioned efficient packing operations, our multiplexer construction works as follows: it guesses the selected \(\ell\)-bit vector, then computes the integers corresponding to the input \(\ell\)-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 \(a\), also the corresponding (densely-packed) integer \(\sum_i 2^{i-1}a_i\).

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 \(C\). Conceptually the three main parts of \(C_{TF}\) are:

  - **Argument decoding**. First, \(C_{TF}\) needs to decode the arguments for the actual instruction that is going to be executed. This involves opcode decoding, register-file multiplexing and so on.

  - **Instruction execution**. After \(C_{TF}\) has determined all arguments for the instruction, we proceed to actually executing the non-memory operations (a result for a memory operation is supplied by the prover as a non-deterministic input to the \(C_{TF}\)). As the underlying proof system only supports rank-1 quadratic equations, it turns out to be more efficient to execute all instructions, than to perform conditional instruction (the latter would raise the polynomial degree, and therefore double the number of gates).

  - **Consistency enforcement**. This step ensures the correct computation of the instruction output (i.e. that output value for the currently needed instruction is correctly fetched from the vector of the results of all instructions, that were executed above), the correct advancement of the program counter, etc.

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 (or, more precisely, a circuit verifying its operation), 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 of an operation, we make sure to have both the binary and integer representation for each argument. Then, each operation in the ALU uses whichever representation is more efficient to use. For instance, bitwise AND, OR, XOR, and NOT are computed using binary representations. In contrast, we use integer representations to compute result and overflow information for addition, subtraction, and multiplication with only $2W$, $2W$, and $3W$ bilinear gates, respectively. For division, we nondeterministically guess the result and verify it with a multiplication. Each time an operation uses integer representations, the output integer can be “unpacked” into its binary representation, via nondeterministic advice. By carefully studying each operation, we obtain a (non-deterministic) circuit for verifying the ALU that, with word size $W = 16$, has merely 343 gates.

Given efficient implementations of multiplexing and the ALU, it is possible to obtain an efficient implementation of $C_{TF}$.

Table 5.1 below shows the number of gates in our implementation of $C_{TF}$ for $|P| \in \{10, 10^2, 10^3\}$, $W \in \{8, 16, 32\}$ and $K \in \{8, 16, 32\}$.

| $|P|$ = 10/100/1000 | $W = 8$ | $W = 16$ | $W = 32$ | $W = 64$ |
|------------------|---------|---------|---------|---------|
| $K = 8$          | 482 / 572 / 1472 | 619 / 709 / 1609 | 892 / 982 / 1882 | 1437 / 1527 / 2427 |
| $K = 16$         | 558 / 648 / 1548 | 695 / 785 / 1685 | 968 / 1058 / 1958 | 1513 / 1603 / 2503 |
| $K = 32$         | 706 / 796 / 1696 | 843 / 933 / 1833 | 1116 / 1206 / 2106 | 1661 / 1751 / 2651 |
| $K = 64$         | 998 / 1088 / 1988 | 1135 / 1225 / 2125 | 1408 / 1498 / 2398 | 1953 / 2043 / 2943 |

Table 5.1: Number of gates in $C_{TF}$ as a function of $W$ and $K$, for different sizes of program $P$.

- **Designing the memory consistency circuit $C_{MC}$.** The predicate $\Pi_{MC}$ is not as complex as the transition function $\Pi_{TF}$, but it is still important to design a small circuit $C_{MC}$ for it. A crucial optimization is afforded by the fact that $\Pi_{MC}$ only cares about the memory address being accessed, and the value loaded/stored, in each state. Deriving these values from the state requires instruction parsing and register-file multiplexing, but it turns out they have already been computed on “the other side” of the routing network, by $C_{TF}$, when verifying code consistency. Thus, we change the routing network (discussed below) to route only such pairs of address and value; there remains for $\Pi_{MC}$ merely to check a simple ordering condition on this.

The predicate $\Pi_{MC}$ is not as complex as the transition function $\Pi_{TF}$, but it is still important to design a small circuit $C_{MC}$ for it. The bottleneck in the computation of $\Pi_{MC}$ is again multiplexing, this time for fetching the two arguments of a memory operation (i.e., the memory address and the value to be loaded/stored). Thus, the natural approach here would be to use additional copies of our efficient multiplexer circuit. Instead, we avoid additional multiplexing altogether by relying on certain intermediate computations from $C_{TF}$.
We thereby obtain a circuit $C_{MC}$ that only contains two integer comparisons and few other logical operations. For instance, when $W = 16$, $C_{MC}$ consists of just 64 gates.

- **Checking routing constraints.** Asymptotically, the routing constraints on the routing network are the most expensive sub-circuit of $C$: there are $\Theta(T \log T)$ nodes in the routing network, compared to $T$ copies of $C_{TF}$ and $C_{MC}$ each. It is thus crucial to check these constraints as efficiently as possible. As discussed for $\Pi_{MC}$, it suffices to route packets consisting of just $2W$ bits (obtained from intermediate computations of $C_{TF}$), instead of whole TinyRAM states. This leads to another important optimization: now that a packet is small, we can pack a whole packet into a single field element (in our typical parameters, $|\mathbb{F}| \geq 2^{2W}$); 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 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. This holds since the size of $\mathbb{F}$ must be $\omega(T)$ for cryptographic reasons.

There are numerous additional details that go into our final construction of the verification circuit $C$. (For instance, another asymptotically-significant component, contributing an additional $2 \cdot T \cdot \log T$ gates, originates from $2T$ integer comparisons on $\log T$-bit integers.) The eventual circuit sizes are as follows, fixing for concreteness a word size $W = 16$, number of registers $K = 16$, and a program length $|P|$ of 100 instructions. The size of $C$ grows with $T$ (when $T$ is a power of 2) as follows:

$$|C| = 4 \cdot T \cdot \log T + 892 \cdot T + 37.$$

In particular, for $\log T < 20$, every cycle of TinyRAM computation costs $\leq 972$ gates to verify. Note that, while the gate count per cycle increases as $T$ increases (since 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.

See Table 5.2 for values of $|C|/T$ for $\log T = \{10, \ldots, 20\}$.

Our profiling shows that $C_{TF}$ comprises roughly 84% of all gates (36% of which are in ALU, 16% are in consistency enforcers, 19% are in argument decoders and 14% to the instruction fetch). Predicate $\Pi_{MC}$ comprises roughly 7% of all gates and approximately the same size was occupied by the routing network. As an interesting side note, 34% of all constraints were devoted to checking Booleanity of various variables; the most expensive sub-circuits of the ALU were those handling shifts and multiplication.

From a software engineering point of view, we tackled the construction of the verification circuit $C$ by developing a library of circuit gadgets, along with functions for composing these gadgets in a clean and modular way. This simplifies future modifications to our circuit generator for application-dependent extensions and customizations, such as supporting other instruction sets, other memory addressing modes, and so on.
| $T$  | $|C|/T$ |
|-----|-------|
| $2^{10}$ | 932.04 |
| $2^{11}$ | 936.02 |
| $2^{12}$ | 940.01 |
| $2^{13}$ | 944.00 |
| $2^{14}$ | 948.00 |
| $2^{15}$ | 952.00 |
| $2^{16}$ | 956.00 |
| $2^{17}$ | 960.00 |
| $2^{18}$ | 964.00 |
| $2^{19}$ | 968.00 |
| $2^{20}$ | 972.00 |

Table 5.2: Number of gates per TinyRAM cycle, with $|P| = 100$, $W = 16$, and $K = 16$.

**Witness map.** Thus far, we have focused on achieving soundness: verifying the validity of an execution trace of a TinyRAM program $P$ by using the circuit $C := \text{circ}(P, T, n)$ output by the circuit generator $\text{circ}$. The circuit generator is run by the key generator when computing the public parameters. (See Figure 6-5.)

Let us now turn to completeness: we need to implement a witness map $\text{wit}(P, T, x, w)$ that computes a satisfying assignment $a$ for $C(x, \cdot)$, whenever $P(x, w)$ accepts in $T$ steps. The witness map is run by the prover when generating a proof. (See Figure 6-6.)

The witness map $\text{wit}$ consists of two main steps.

1. **From inputs to execution trace.** We implemented a TinyRAM simulator, denoted $\text{Simulate}$, that, given as input $(P, T, x, w)$, outputs the $T$-step execution trace $\text{tr}$ of $P$ on inputs $(x, w)$. This step is conceptually straightforward: it is implemented as a fetch-and-execute loop written in a high-level language, C++. Unlike the subsequent steps, it does not involve circuit representation. Performance of the simulation inessential, since, in the full prover, running time is dominated by subsequent steps.

2. **From execution trace to satisfying assignment.** We implemented a function, denoted $\text{ExtendAndRoute}$, that, given a valid $T$-step trace $\text{tr}$ for $P(x, w)$, outputs a satisfying assignment $a$ for $C(x, \cdot)$. Computing $a$ involves several sub-steps, corresponding to finding suitable satisfying assignments to the different sub-circuits of $C$, as we now describe.

   - The first task is to deduce from $\text{tr}$ a satisfying assignment for each copy of $C_{TF}$ in $C$. In order to satisfy the $i$-th copy of $C_{TF}$, it is not enough to provide the $i$-th and $(i + 1)$-th line in the trace $\text{tr}$ as input to the $i$-th copy of $C_{TF}$. Indeed, each copy of $C_{TF}$ also expects nondeterministic advice. (For instance, each multiplexer in $C_{TF}$ expects auxiliary advice; and so do many sub-circuits of the ALU, such as the sub-circuit responsible for verifying a division’s result.) Thus, in this step we compute the necessary auxiliary advice for each copy of $C_{TF}$.

   - The second task is to deduce from $\text{tr}$ a satisfying assignment for each copy of $C_{MC}$ in $C$. We do so by first stable-sorting the trace $\text{tr}$, by address accessed, in order
to obtain $\text{MemSort}(\text{tr})$. Then, we provide the $i$-th and $(i + 1)$-th line in the trace $\text{MemSort}(\text{tr})$ as input to the $i$-th copy of $C_{MC}$. Similarly as before, each copy of $C_{MC}$ also requires various nondeterministic advice, mostly consisting of internal wire values of computations of the corresponding copy of $C_{TF}$.

- The third and final task is to deduce a satisfying assignment to the sub-circuit of $C$ responsible for checking that $\text{MemSort}(\text{tr})$ is a suitable sorting of $\text{tr}$. To do so, we deduce from $\text{tr}$ and $\text{MemSort}(\text{tr})$ the permutation $\pi$ that we need to route on the Beneš network; we then compute the switch settings for the network by using a standard routing algorithm [Ben65, Wak68, OTW71, Lei92, NS82]; from these switch settings and $\text{tr}$, we can then deduce the satisfying assignment. Asymptotically, this is the most expensive part of the witness map.

The above concludes the description of our witness map $\text{wit}$. 

52
Chapter 6

Experimental evaluation

In this chapter we discuss the measured performance of our system. Plugging our linear PCP for arithmetic circuits (Section 3.4) into the transformation (Section 3.3), we obtain an implementation of zk-SNARKs for arithmetic circuit satisfiability with essentially-optimal asymptotic efficiency: linear-time key generator, quasilinear-time prover, and linear-time verifier.

6.1 Instantiation of $G_1, G_2, G_T$ and pairing

We begin by describing the algebraic setup that provides a concrete instantiation of the prime-order groups $G_1$ and $G_2$.

We consider an Edwards curve $E$ defined over the field $\mathbb{F}_q$ where $q$ is a prime of 183 bits. The curve group $E(\mathbb{F}_q)$ has order $4r$ where $r$ is a prime of 181 bits. In particular, the value $\rho = \frac{\log q}{\log r}$ is approximately 1.011. Moreover, $r - 1$ is divisible by $2^{30}$.

The group $G_1$ is a cyclic subgroup of $E(\mathbb{F}_q)$ of order $r$, and $G_2$ is a cyclic subgroup of $E'(\mathbb{F}_{q^6})$ of order $r$ where $E'$ is a quadratic twist of $E$ (thus $E'$ is a twisted Edwards curve [BBJLP08]); this instantiation provides 80 bits of security [FST10]. (In particular, the security parameter is now implicit and we thus omit it as an explicit input to the SNARK key generator in the discussion below.) Thus, each $G_1$ group element (when compressed) is 184 bits; each $G_2$ group element (when compressed) is 550 bits.

The curve $E$ was found, after $\approx 2^{38}$ trials, by following the strategy outlined in Section 3.3. The same strategy can be used, with more trials, to find curves where $r - 1$ is divisible by a larger power of 2: roughly $\Omega(2^\ell)$ trials are needed to find a choice of parameters where $2^\ell$ divides $r - 1$.

Taking the “target” group $G_T$ to be a suitable subgroup of $\mathbb{F}_{q^6}^*$ (since the embedding degree of $E$ is 6), we choose the non-degenerate bilinear map $e: G_1 \times G_2 \to G_T$ to be the (reduced) Tate pairing [FR94, FMR06].
6.2 Performance of our zk-SNARK for circuit satisfiability

Next, we discuss the concrete performance of the generator, prover, and verifier. Our experiments were running as single-threaded code on a 2.4 GHz Intel E7-8870 CPU with 256 GB of RAM. (While our prototype does not exploit parallelism, our algorithms are highly parallelizable, so that latency can be greatly improved.)

6.2.1 Performance of key generation

Given an arithmetic circuit $C: \mathbb{F}^n \times \mathbb{F}^h \rightarrow \mathbb{F}$ as input (where $\mathbb{F} = \mathbb{F}_r$), the SNARK key generator $G$ outputs:

- a proving key $\sigma$ of $(12|C| + 2n + 40)$ group elements from $\mathbb{G}_1$ and $|C|$ from $\mathbb{G}_2$ (see Figure 6-1 (left)); and

- a verification key $\tau$ of $(n + 2)$ group elements from $\mathbb{G}_1$ and 6 from $\mathbb{G}_2$ (see Figure 6-1 (right)).

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 field and elliptic-curve groups; storing these public parameters only requires < 4000 bits.

Figure 6-2 shows the measured number of operations (in $\mathbb{F}_r, \mathbb{G}_1, \mathbb{G}_2$) and running time of $G(C)$ as a function of $|C|$ (for some fixed value of $n$).

For instance, when $|C| \approx 2 \cdot 10^6$, $G$ terminates in less than 20 minutes.
Figure 6-2: Number of operations in $\mathbb{F}_r$, $\mathbb{G}_1$, $\mathbb{G}_2$ (left) and running time (right) of the SNARK key generator $G(C)$ as a function of $|C|$, the number of gates in $C$. Also shown (right) is the time spent by $G$ just for sampling linear queries; the difference is spent on cryptographic operations for encoding these queries. As expected, the asymptotic dependence on $C$ is $O(|C|)$. In both graphs, we can see the effect of building multi-exponentiation tables for $\mathbb{G}_1$ and $\mathbb{G}_2$, as the substantial constant term that initially dwarfs the “main” linear work done by the prover; without building multi-exponentiation tables graphs for $\mathbb{G}_1$ and $\mathbb{G}_2$ operations would closely mirror graph of $F_p$ operations, albeit with a higher linear factor.

6.2.2 Performance of proving

Given $\sigma$ and $(x, a)$ in the relation $\mathcal{R}_C$, the SNARK prover outputs a proof consisting of 12 group elements (11 in $\mathbb{G}_1$ and 1 in $\mathbb{G}_2$). The proof length is 2576 bits (322 bytes). For comparison, we can fit 4 proofs into a single TCP packet, which is 1460 bytes.

Figure 6-3 shows the measured number of operations (in $\mathbb{F}_r$, $\mathbb{G}_1$, $\mathbb{G}_2$) and running time of $P(\sigma, x, a)$ as a function of $|C|$ (for some fixed value of $n$).

For instance, when $|C| \approx 2 \cdot 10^6$, $P$ terminates in less than 22 minutes.

6.2.3 Performance of verifying

Given $\tau$, an input $x$, and a proof $\pi$, the SNARK verifier computes the decision bit. To do so, the verifier evaluates 21 pairings and solves a multi-exponentiation problem of size $|x|$.$^1$

Figure 6-4 shows the measured running time of $V(\tau, x, \pi)$ as a function of $|x|$. For instance:

- when $|x| \leq 2^6$, $V$ terminates in less than 103 milliseconds;
- when $|x| \leq 2^{17}$, $V$ terminates in less than 4.68 seconds.

---

$^1$ If $x$ is a vector of field elements in $\mathbb{F}_r$, then $|x|$ is the number of elements in the vector. If $x$ is a bit string, then we can take $|x|$ to be the number of bits in $x$ divided by 181, because we can “pack” 181 bits into a single field element of $\mathbb{F}_r$. 

55
Figure 6-3: Number of operations $F_r, G_1, G_2$ (left) and running time (right) of the SNARK prover $P(\sigma, x, a)$ as a function of $|C|$, the number of gates in $C$. Also shown (right) is the (quasilinear) time spent by $P$ just for computing the linear PCP proof; the difference is the (linear) time spent on cryptographic operations for homomorphically evaluating query answers. As expected, the asymptotic dependence on $C$ is $O(|C| \log |C|)$. Moreover, for small values of $|C|$ the (linear) cryptographic overhead dominates; as $|C|$ increases, the (quasilinear) computation of the linear PCP proof eventually will dominate.

Figure 6-4: Running time of the SNARK verifier $V(\tau, x, \pi)$ as a function of the number of bits in the input $x$. Also shown is the (linear) time spent by $V$ just for the checking input consistency; the difference is for computing a constant number of pairings (21 in total). Recall that the input to a circuit is generally much smaller than the circuit’s size, so the input size should be thought of as relatively small. As expected, the dependence on $|x|$ is $O(|x|)$. 

56
We emphasize that the above performance holds no matter how large is the circuit $C$.

6.3 Performance of our zk-SNARK for arbitrary C programs

In Section 4.3 we discussed our reduction from the correctness of program execution to circuit satisfiability and then in Section 6.2 we discussed our zk-SNARK for circuit satisfiability. In this section, we discuss the performance of the system obtained by combining these two components: an implementation of a zk-SNARK for the correctness of program executions. This system provides a solution for non-interactively delegating arbitrary $\mathcal{NP}$ 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 system.

6.3.1 System overview

The zk-SNARK for the correctness of program executions consists of three algorithms ($G^*, P^*, V^*$):

- The key generator $G^*$, given a TinyRAM program $P$, input size $n$, and time bound $T$, outputs a proving key $\sigma$ and a verification key $\tau$ that can be used to (respectively) prove and verify $T$-step computations of $P$ on primary inputs of $n$ words. (The auxiliary input may be longer than $n$ words.)

- The prover $P^*$, given the proving key $\sigma$, the TinyRAM program $P$, a $n$-word primary input $x$, time bound $T$, and auxiliary input $w$, outputs a proof $\pi$, attesting to the fact that $P(x, w)$ accepts in $T$ steps.

- The verifier $V^*$, given the verification key $\tau$, a $n$-word primary input $x$, and proof $\pi$, checks whether $P(x, w)$ accepts in $T$ steps for some choice of auxiliary input $w$.

Our GCC-based compiler can be used to obtain TinyRAM programs from C programs. See Figure 6-5 and Figure 6-6 below for an overview of how the three algorithms ($G^*, P^*, V^*$) are obtained from $(\text{circ, wit, wit}^{-1})$, which is our circuit reduction, and $(G, P, V)$, which is our zk-SNARK for circuit satisfiability.

6.3.2 System performance

We now discuss the performance of our system, starting with the efficiency of compilation.

- **Compiling.** The important efficiency measures of compiling from C code to TinyRAM assembly are code size (the number of instructions in the generated assembly code) and execution time (the number of machine steps needed to execute the assembly code). As

---

2 As in Section 3.5, we conducted these experiments using single-threaded code running on a 2.4 GHz Intel E7-8870 CPU with 256 GB of RAM.
Offline Phase (once)

TinyRAM program time bound primary input size

\( P \)
\( T \)
\( n \)

\( \sigma \)
\( \tau \)

\( G \)
\( G^* \)

Circuit generator

\( \mathcal{C} \)

Proving key

Verification key

Online Phase (any number of times)

TinyRAM program time bound primary input proving key

\( P \)
\( T \)
\( x \)
\( \sigma \)

\( \omega \)

Witness map

Simulate

Extend & Route

\( a \)

Assignment

\( \pi \)

Proof

Zk SNARK key generator

\( G \)

Zk SNARK prover

\( V \)

Zk SNARK verifier

0/1

Figure 6-5: Overview of offline phase.

Figure 6-6: Overview of online phase.
discussed in Section 4.2, initial experiments indicate that both code size and execution
time for our compiler are only a small multiplicative factor greater than those incurred
when compiling to other architectures (such as x86, ARM, or AVR) by relying on
existing compilers.

Next, we discuss the performance of our implementation of \((G^*, P^*, V^*)\). For concreteness,
we fix word size \(W = 16\) and number of registers \(K = 16\).

- **Key generation.** The efficiency of the key generator \(G^*(P, n, T)\) essentially only de-
  pends on the number of instructions in the program \(P\), the input size \(n\), and the time
  bound \(T\). Fixing any 100-instruction program \(P\), and input size \(n = 100\), we study
  the efficiency of \(G^*(P, n, T)\) as the time bound \(T\) grows. Specifically, the graphs in
  Figure 6-7 show, as a function of \(T\), the running time of \(G^*\), the number of gates in
  the circuit \(C\) (generated by \(G^*\) as an intermediate value), and the number of group
  elements in the proving key \(\sigma\) output by \(G^*\). (We do not plot the number of group
  elements in the verification key \(\tau\) output by \(G^*\), because the verification key always
  has \(n + 2\) elements from \(G_1\) and 6 elements from \(G_2\), regardless of the value of
  \(T\).)

- **Proving.** The efficiency of the prover \(P^*(\sigma, P, x, T, w)\) essentially only depends on the
  number of instructions in the program \(P\), the input size \(n\) (i.e., the number of words
  in \(x\)), and the time bound \(T\). Fixing any 100-instruction program \(P\), primary input \(x\)
  with \(n = 100\) words, and auxiliary input \(w\), we study the efficiency of \(P^*(\sigma, P, x, T, w)\)
  as the time bound \(T\) grows. Specifically, in Figure 6-8 on page 60, we plot the running
time of \(P^*\) as a function of \(T\). Recall that the proof generated by \(P\) always consist of
12 group elements (regardless of the value of \(T\), or other inputs)

- **Verifying.** The efficiency of the verifier \(V^*(\tau, x, x)\) essentially only depends on the input
  size \(n\) (i.e., the number of words in \(x\)). Indeed, the verifier \(V^*\) receives a verification key
  \(\tau\), primary input \(x\), and proof \(\pi\), and then invokes \(V\) on these inputs. More precisely,
  \(V^*(\tau, x, \pi)\) actually coincides with \(V(\tau, x', \pi)\) where \(x' \neq x\) is a string of \(2W(n + 1) + 1\)
  bits obtained via a deterministic mapping applied to \(x\). (Note that a primary input \(x\)
  of with \(n\) words contains \(Wn\) bits.) The mapping from \(x\) to \(x'\) arises from a technicality
  for correctly performing input consistency.

Note that the circuit reduction was already performed by \(G^*\) and is implicit in the
verification key \(\tau\); thus \(V^*\) itself is oblivious to the circuit reduction (up to the already
mentioned technicality about mapping \(x\) to \(x'\)).

The performance of \(V\) as a function of the number of bits in the input was discussed
in Section 3.5, and specifically Figure 6-4. So we do not produce any new graphs for
\(V^*\).

### 6.3.3 Performance for Rectilinear TSP Example

We now report the system’s performance when used for an illustrative example. The example
proves and verifies claims of membership in the rectilinear Traveling Salesman Problem
(rectilinear TSP) language, defined as follows.
Figure 6-7: On the left: running time of $G^*$ as a function of the time bound $T$; also shown is the running time of the circuit generator $\text{circ}$, while the remaining time is spent running $G$. (As $T$ increases, the running time of $G$ dominates the running time of $G^*$.) In the center: number of gates in the circuit $C$, output by the circuit generator $\text{circ}$, as a function of $T$. On the right: number of $G_1$ and $G_2$ group elements in the proving key output by $G^*$ as a function of $T$.

Figure 6-8: Running time of $P^*$ as a function of the time bound $T$. Also shown is the running time of the witness map $\text{wit}$; the remaining time is spent running $P$. As $T$ increases, the running time of $P$ dominates the running time of $P^*$.

A complete weighted graph $G$ is specified by a list $((x_1, y_1), \ldots, (x_n, y_n))$ where $x_i, y_i \in \mathbb{Z}$; the $i$-th pair $(x_i, y_i)$ specifies the coordinates in $\mathbb{Z} \times \mathbb{Z}$ of the $i$-th vertex $v_i$; any two vertices $v_i$ and $v_j$ have weight $\Delta(v_i, v_j)$ given by the Manhattan distance (in $\mathbb{Z} \times \mathbb{Z}$) between $v_i$ and $v_j$.

Given a complete weighted graph $G$, the rectilinear TSP language, $L_G$, is the language of pairs $(s, B)$, where $s$ is a source vertex in $G$ and $B \in \mathbb{Z}_+$ is a weight bound, such that there exists a Hamiltonian path $p$ in $G$ starting at $s$ and with total weight at most $B$.

We chose above rectilinear TSP language in order to exercise various features of our system:
• Rectilinear TSP is \( \text{NP} \)-complete (when considered across all graphs \( G \)) [EKP85], so the proof-of-knowledge property gives a non-trivial guarantee. Namely, proving knowledge of a Hamiltonian path of low weight — when such a path is supposedly hard to find — is quite meaningful. (While proof of knowledge does tend to be more useful in cryptographic examples, we opted for a more familiar “classical” NP example.)

• Valid witnesses are often not unique, so that the zero-knowledge property gives a non-trivial guarantee.

• It is easy to write a linear-time C program that decides the relation for \( L_G \): given an instance \((s, B)\) and a candidate witness \( p \), the program checks that \( p \) is a valid path in \( G \) and that its total weight is at most \( B \).

In contrast, there may not exist a linear-size circuit for this task: even a quasilinear-size circuit seems to require routing/sorting techniques (similar to those used in our circuit generator; cf. Section 4.3). Thus, while simple, deciding \( L_G \) efficiently makes crucial use of random access to memory.

• Instances in the language \( L_G \) are much smaller than witnesses: a single vertex and an integer bound, vs. a Hamiltonian path. Hence, the time to verify a proof (which is linear in the instance size) will be much smaller than simply running the aforementioned program to decide the language (given the full witness).

Fixing a 200-node graph \( G \) (as mentioned above, fixing a set 200 of points in plane fixes the (complete) graph of edges, where each edge has weight equal to the distance between the points), we obtained the following.

• **Compiling.** We wrote a simple C implementation of an NP decider for \( L_G \) (so that the description of \( G \) is hardcoded in the decider), and compiled it to TinyRAM using our GCC-based compiler. The resulting TinyRAM program \( P \) consists of 1105 instructions. When running \( P \) on a primary input \( x = (s, B) \) and auxiliary input \( w = p \) (specified as a list of vertices), where \( s = v_0 \) and \( p \) is a specific Hamiltonian path in \( G \) of total weight at most \( B \), \( P \) accepts after 11 001 steps.

• **Key generation.** Running the key generator \( G^* \) took 247 minutes. The generator \( G^* \) produced a proving key with 392153579 group elements in \( G_1 \) and 36847976 group elements in \( G_2 \); and a verification key with 9 group elements in \( G_1 \) and 6 group elements in \( G_2 \). Of the total time, 307 seconds were spent in evaluating the circuit generator \( \text{circ} \) to compute \( C \) and the remaining time was spent in running \( G \) on \( C \). The circuit \( C \) (which is an intermediate value of \( G^* \)’s computation) consisted of 32047142 gates.

• **Proving.** Running the prover \( P^* \) took 155 minutes, and produced a proof of 12 group elements (11 in \( G_1 \) and 1 in \( G_2 \)). Of the total time, 318 seconds were spent in evaluating the witness map \( \text{wit} \), which outputs a satisfying assignment \( a \) for \( C \), and the remaining time was spent in running \( P \) on \((\sigma, x, a)\) to compute the proof.

• **Verifying.** Running the verifier \( V^* \) took 0.11 seconds. Essentially this entire time was spent running \( V \).
Straightforward optimizations will significantly improve the above running times, as discussed next.

### 6.4 Future optimizations

Our system is a proof-of-concept prototype, not an industrial-grade implementation. In particular, we did not put effort into any “second-order” optimizations that are standard and well-understood, but instead have focused our effort on optimizations that are novel to our work (and are thus less understood), e.g., optimizing the number of gates in the circuit checking TinyRAM’s transition function.

Nonetheless, for completeness, we briefly mention several standard optimizations that will significantly improve the efficiency and scalability of our prototype.

- **Parallelization.** Essentially all the computations required of the generator, prover, and verifier can be parallelized. In particular, routing on Beneš networks, sorting, polynomial interpolation/evaluation, multi-exponentiation, and others — all of these are highly-parallelizable (i.e., have polylogarithmic-depth circuits). Parallel implementations of all of these computational tasks are well-studied, and it should not be difficult to make our prototype leverage all available cores so to significantly reduce latency.

- **Computing in blocks.** Most of the computational problems mentioned in the previous paragraph achieve excellent time complexity at the cost of large space complexity. The large space complexity poses a serious obstacle to the scalability of the zk-SNARK to lengthy TinyRAM computations.

  For example, the FFT algorithm improves over naive interpolation/evaluation, but does so at the cost of requiring random-linear space (with random access). As long as this fits into available RAM, (quasilinear) FFT is faster than native (quadratic) interpolation/evaluation. But for large problem sizes (e.g., such as those arising when proving correctness of lengthy computations), RAM becomes a bottleneck. This problem is traditionally mitigated by adopting a hybrid approach: the interpolation/evaluation problem is divided into smaller sub-problems (the “blocks”), the FFT algorithm is used to solve (separately) each smaller subproblem, and the solutions to the subproblems are combined using a naive algorithm. Doing so increases time complexity, but decreases space complexity. The choice of the block size allows one to tailor the resulting computational problem to the hardware carrying out the computation.

  Similar ideas apply to mitigating the space complexity of routing on Beneš networks, multi-exponentiation, etc. Applying these will improve scalability of the implementation, by removing the memory bottleneck.

- **Optimized field arithmetic.** Essentially all of the computations of the algorithms of \((G, P, V)\) — the zk-SNARK for circuit satisfiability — consist of field operations over a large prime field. In particular, tightly optimizing arithmetic for such fields (by

\[3\]With one exception: the actual execution of the TinyRAM program by the prover, in order to generate the execution trace, cannot of course be parallelized (in the general case).
taking into account the specific architecture at hand, the specific field that is used, etc.) has a significant impact on the efficiency of these three algorithms. Of course, optimizing arithmetic for large prime fields is a well-studied problem (e.g., it arises in many cryptographic applications). Thus, it should not be difficult to improve the running times that we measured in Section 3.5. (For this reason the number of field operations is arguably a more important measure of efficiency, and in Section 3.5 we do report the number of field operations alongside running times.)
Chapter 7

Conclusion

In this work we have described a proof system that allows efficient verification of \( \text{NP} \) statements, given proofs produced by an untrusted yet computationally-bounded prover. Our system is publicly verifiable: after a trusted third-party has generated a proving key and a verification key, anyone can use the proving key to generate non-interactive proofs for adaptively-chosen \( \text{NP} \) statements, and the proofs can be verified by anyone using the verification key. Moreover, our system is statistically zero-knowledge and the generated public parameters are reusable.

Our system directly supports a correct executions of programs on TinyRAM, a random-access machine tailored for efficient verification of nondeterministic computations we design. Together with TinyRAM port of gcc compiler this achieves the first practical realization of a zk-SNARKs for program executions, in the preprocessing model. This cryptographic primitive is a powerful solution for delegating \( \text{NP} \) computations, and enjoys many features not achieved by primitives implemented in prior works, most importantly, succinct verification and support for arbitrary computations.

7.1 Future work

We would like to outline directions for future work to achieve succinct verification in its full generality.

SNARKs without pre-processing. Despite recent theoretical and practical advances, the “Holy Grail” of verified computation — succinct verification without pre-processing — remains sought after and is subject of further work.

Oblivious key generation. The two drawbacks associated with pre-processing SNARKs are: (a) costly pre-processing phase to be performed before any proofs can be verified, and (b) the requirement for the generator to be trusted by both the prover and the verifier. In some applications the latter poses a greater challenge than the former: verifier might have enough computational resources to do “one-shot” pre-computation, but it might be hard to find a third party that the prover and the verifier both trust. If SNARKs without pre-processing are too costly to achieve in practice, can the keypair be generated in public coin
setting, thereby alleviating the trust requirement? We have preliminary results that show this should indeed be possible under stronger (but plausible) complexity assumptions.
Appendix A

Examples Used in Section 4.2

The graphs in Figure 4-2 and Figure 4-3 refer to specific examples of C code that we wrote in order to obtain those benchmarks. We briefly describe the programs that we wrote. We selected a set of simple, natural examples that demonstrate various program styles in C; these examples exercise memory accesses, integer arithmetic, and logical calculations.

1. **Pointer chasing.** Our example takes as input a permutation \( \pi \) on a domain \( \{1, \ldots, w\} \) and an integer \( d \) (we use \( d = 3 \)), and computes \( \pi^d \), i.e., the composition of \( \pi \) with itself \( d \) times. This example exercises random accesses: while random access machines can compute \( \pi^d \) in \( O(d \cdot w) \) time, a (naive) arithmetic circuits for this function, using a \( w \)-to-1 multiplexer for choosing each element, has \( \tilde{O}(d \cdot w^2) \) size.

2. **Game of Life.** Conway’s *Game of Life* [Gar70] is a cellular automaton on an \( m \times m \) mesh where each cell is initialized as either dead or alive. The game transitions from a generation to the next; in every such transition, each cell either dies or become alive depending on the number of alive neighbors it has. Fixing \( m = 5 \), our example program checks whether, given an initial configuration, a target configuration, and a positive integer \( d \), the target configuration is the result of simulating Game of Life for \( d \) generations starting from the initial configuration.

3. **Matrix multiplication.** Our example program takes as input two square integer matrices (of the same dimension), and multiplies them using the naive matrix-multiplication algorithm.

4. **Polynomial evaluation.** Our example program takes as input a polynomial of some degree \( d \) and \( k = d + 2 \) points, and evaluates the polynomial at each of the \( k \) points using the naive algorithm. Both the coefficients and points are 16-bit integers.

5. **Single-source shortest paths.** A single-source shortest path problem is specified by a weighted graph \( G = (V, E) \) and a source node \( s \in V \). The goal is to find the shortest

---

1 A more efficient circuit implementation would represent the permutations using routing networks (with nondeterministically-chosen switch setting), and achieve a circuit of size \( O(dw \log w) \). The circuit generator that we implemented does this implicitly using its handling of random access to memory (following [BCGT13a]), which supports arbitrary read/write access patterns.
path between $s$ and every node in $G$. Our example takes as input a positively-weighted graph $G$ (where nodes have in-degree 20) and a source $s$, and finds the shortest path from $s$ to all the nodes in $G$ using Dijkstra’s algorithm [Dij59].

6. **RC4 stream cipher.** The RC4 stream cipher [Gol97] maintains a 256-byte state, which it repeatedly updates during the initial key scheduling algorithm (KSA) and, subsequently, during the repeated invocation of the pseudorandomness generation algorithm (PRGA). Every invocation of the PRGA produces a pseudorandom sequence. Our example program takes as input an RC4 secret key, a positive integer $d$, and a “target” 128-bit string $t$; then, the program initializes the KSA with the secret key, produces a stream of $d$ pseudorandom bytes using the PRGA, and finally checks that the 128-bit suffix of the resulting stream equals to $t$. 
Bibliography

[23a] 23andMe, Inc. 23andMe Research. URL: https://www.23andme.com/research/.


[BCCT13b] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. On the concrete effi-


