Computational integrity with a public random string from quasi-linear PCPs.

Eli Ben-Sasson[1], Iddo Bentov[2], Alessandro Chiesa[3], Ariel Gabizon[4], Daniel Genkin[5], Matan Hamilis[1], Evgenya Pergament[1], Michael Riabzev[1], Mark Silberstein[1], Eran Tromer[6] and Madars Virza[7]

[1] Technion — Israel Institute of Technology
[2] Cornell University[★★★]
[3] University of California, Berkeley
[4] Zerocoin Electric Coin Company (Zcash)[★★★]
[5] University of Pennsylvania and University of Maryland, College Park[†]
[6] Tel Aviv University
[7] Massachusetts Institute of Technology

A party executing a computation on behalf of others may benefit from misreporting its output. Cryptographic protocols that detect this can facilitate decentralized systems with stringent computational integrity requirements. For the computation's result to be publicly trustworthy, it is moreover imperative to use publicly verifiable protocols that have no "backdoors" or secret keys that enable forgery.

Probabilistically Checkable Proof (PCP) **systems can be used to construct such protocols, but some of the main components of such systems — *proof composition* and low-degree testing via *PCPs of Proximity (PCPPs)* — have been considered efficiently only asymptotically, for unrealistically large computations. Recent cryptographic alternatives suffer from a non-public setup phase, or require large verification time.**

This work introduces SCI, the first implementation of a scalable PCP system (that uses both PCPPs and proof composition). We used SCI to prove correctness of executions of up to 2^{20} cycles of a simple processor, and calculated its *break-even point*: the minimal input size for which naïve verification via re-execution becomes more costly than PCP-based verification.

This marks the transition of core PCP techniques (like *proof composition* and *PCPs of Proximity*) from mathematical theory to practical system engineering. The thresholds obtained are nearly achievable and hence show that PCP-supported computational integrity is closer to reality than previously assumed.

[★★★] Work done while at Technion
[†] Work done while at Technion and Tel Aviv University

1 Introduction

Computational Integrity An unobserved party is often required to execute a program $\mathbb{P}$ on data x, using auxiliary data w. Yet, that party might benefit from misreporting the output y. For example:

1. Individuals and companies may benefit financially from reporting lower tax payments; in this case $\mathbb{P}$ is the program that computes tax, x is the tax-relevant data (w is the empty string) and y is the resulting tax.
2. Criminals may benefit if an innocent individual (or no individual) is prosecuted based on faulty crime-scene data analysis, and corrupt law enforcement officials to reach this outcome. In this case $\mathbb{P}$ is the program that analyzes crime-scene data, x may contain the cryptographic hashes of (i) a criminal DNA database and (ii) DNA fingerprints taken from the crime-scene, w is the preimage of (i), (ii) and y would be the name of a suspect.
3. Health-care and other insurance companies may benefit from mis-computing policy rates. In this case $\mathbb{P}$ may be a government-approved program that computes policy rates, x is the identifying number of a patient, w would be her medical history (including, perhaps, her DNA sequence) and y is the policy rate.

Naturally, correctness and integrity of the input data x, w are preliminary requirements for obtaining a correct output y; These inputs often arrives from third parties and can be digitally signed by them, hence changing (x, w) maliciously to (x', w') would require their collusion. Instead, the main focus of this work is on ensuring the integrity of the *computation* $\mathbb{P}$ itself, e.g., ensuring that the reported tax y is correct with respect to the explicit input x, program $\mathbb{P}$ and some auxiliary input w. In spite of incentives to cheat, we often assume that unobserved parties operate with *computational integrity* (CI) meaning that CI statements like

$$\tau_{(\mathbb{P},x,y,T)} := \text{``}\exists w \text{ such that } y = \text{ output of } \mathbb{P} \text{ on inputs } x, w \text{ after } T \text{ steps''} \quad (*)$$

are considered true, even when the party making the statement could benefit from replacing y with $y' \neq y$. The assumption that parties operate with computational integrity is backed by (i) legislation and (ii) regulation, and also relies on (iii) the economic value of "integrity" to individuals, businesses and government. Manual enforcement of CI via audits and reports by trusted third parties is labor-intensive, and yet leaves the door open to corruption of those third parties. Automated CI based on cryptography (also called *delegation of computation* [44], *certified computation* [33] and *verifiable computation* [41]) could potentially replace this manual labor and, more importantly, introduce integrity to settings in which it is currently too costly to achieve.

Interactive proof (IP) systems [5, 45] revolutionized cryptographic CI by initiating an approach that led (see below) to a viable theoretical solution to the problem of discovering false CI statements. In such systems the party that makes the CI statement (*) is represented by a *prover* which is a (randomized)

algorithm. The prover tries to convince a *verifier* — an efficient randomized algorithm — that (*) is true via a court-of-law-style *interactive* protocol in which the verifier "interrogates" the prover over several rounds of communication. The protocol ends with the verifier announcing its verdict which is either to "accept" $\tau_{(\mathbb{P},x,y,T)}$ as true, or to "reject" it. The systems we focus on have only one-sided error: all true statement can be supported by a prover that causes the verifier to accept them but the verifier may err and accept falsities; the probability of error is known as the *soundness-error*.

Probabilistically checkable proof (PCP) systems[8] [4, 3, 2, 1] are a particularly efficient multi-prover interactive proof (MIP) system [8] in terms of the amount of communication between prover and verifier, verification time, the number of rounds of interaction and soundness-error. Assuming T is given in binary, the set of true CI statements (eq:statement) is a **NEXP**-complete language and PCPs are powerful enough to prove membership in this language. Here, the prover writes once a string of bits $\pi_{(\mathbb{P},x,y,T)}$ known as a PCP; its length is polynomial in the execution time T. Total verifier running time is $\operatorname{poly} \log T$, which is (i) negligible compared to the naïve solution of re-executing $\mathbb{P}$ at a cost of T steps and (ii) nearly-optimal because every proof system for general CI statements must have the verifier running time be at least $\Omega(\log T)$. Using a single round, the verifier asks to read a small (randomly selected) number of bits of $\pi_{(\mathbb{P},x,y,T)}$; clearly the verifier cannot read more bits than its running time ($\operatorname{poly} \log T$) allows, and this amount can be further reduced to a small constant that is independent of T (cf. [67, 50, 35, 64]). Initial constructions required proofs of length $\operatorname{poly}(T)$ but length has been reduced since then [49, 43, 24, 21] and state-of-the-art proofs are of *quasi-linear* length in T, i.e., length $T \cdot \operatorname{poly} \log T$ [23, 35, 20, 63] and can be computed in quasi-linear time as well [13]. The system reported — called Scalable Computational Integrity (SCI) — implements the quasi-linear PCP system [23, 13] with certain improvements (described later).

In many cases the prover needs to preserve the privacy of the auxiliary input w (as is the case with examples 2, 3 above) while at the same time proving that it "knows" w, as opposed to merely proving that w exists. Privacy-preserving, or *zero knowledge (ZK) proofs* [45] and *ZK proofs of knowledge* [7] can be constructed from any PCP system in polynomial time [56, 37, 57] (cf. [53, 54, 61, 55]). Certain "algebraic" PCP systems, including SCI, can be converted to ZK proofs of knowledge with only a quasilinear increase in running time [11]; **implementing this enhancement is left to future work.**

A PCP verifier requires random access to bits of $\pi_{(\mathbb{P},x,y,T)}$; a naïve implementation in which prover sends the whole proof to the verifier would cost $\operatorname{poly}(T)$ communication (and verification time) but a collision-resistant hash function can be used to reduce communication and verifier running time to $\operatorname{poly} \log T$ [56]. The three messages transmitted between prover and verifier ((1) prover sends proof; (2) verifier sends queries; (3) prover answers queries) can be reduced to a single message from the prover, if both parties have access to the same random function [62]; this can be realized using a standard cryptographic hash function such as

---

[8] PCPs are also known as *holographic*, and *transparent* proof systems.

SHA-3, via the Fiat-Shamir heuristic [39] (or via an extractable collision resistant hash function [27]). The single message (published by the prover) is known as a *succinct computationally sound* (CS) proof $\hat{\pi}$; its length is $\text{poly} \log T$ and it can now be appended to $\tau_{(\mathbb{P}, x, y, T)}$ and then publicly verified in time $\text{poly} \log T$ with no further interaction with the prover. We refer to $\hat{\pi}$ as a *hash-based (CI) proof* to emphasize that the only cryptographic primitive needed to implement it is a hash function.

Prior CI solutions In spite of the asymptotic efficiency of PCPs, prior CI approaches (recounted below) did not implement a PCP system. To quote from the recent survey [78], the reason for this was that *"the proofs arising from the PCP theorem (despite asymptotic improvements) were so long and complicated that it would have taken thousands of years to generate and check them, and would have needed more storage bits than there are atoms in the universe"*. Due to this view (which this work challenges), five main alternatives have been explored recently, described below. Like SCI, all rely on *arithmetization* [60], the reduction of computational integrity statements (*) to systems of low-degree polynomials over finite fields. But in contrast to SCI, all previous solutions circumvent the use of core PCP techniques like *proof composition* [2], *low-degree testing* and the use of *PCPs of proximity (PCPP)* [20, 36]; these techniques are crucial for obtaining *succinctly verifiable* proofs with a *public setup* process, which SCI is the first to implement.

IP-based: The *proofs for muggles* approach [44] scales down Interactive Proofs (IP) from **PSPACE** to **P** and leads to excellent solutions for a limited yet interesting class of programs: those with high parallelism and small memory consumption; prover time for *IP-based* systems was reduced to quasi-linear [34] and implemented in a number of works [33, 74, 76].

LPCP-based: [52] proposed using additively homomorphic encryption (AHE) and *linear PCPs (LPCP)* to build CI proof systems that are interactive, and where the verifier's work is amortized over multiple statements; cf. [70, 73, 72] for implementations of *LPCP-based* systems.

KOE-based: A sequence of works [47, 41, 59, 29, 42] improved on [52] by relying on *Knowledge Of Exponent (KOE)* assumptions and bilinear pairings over elliptic curves. *KOE-based* systems were implemented in [66, 71, 15, 19, 77], and further optimizations of this latter system for specific applications related to Bitcoin [65] such as smart contracts [58] and anonymous payment systems [12] are already being evaluated by commercial entities [46].

IVC-based: KOE-based systems require a proving key k_P (discussed below) that is longer than T, the number of computation cycles. Incrementally verifiable computation (IVC) [75] and bootstrapping [28] shorten the length of k_P to $\text{poly} \log T$ and an *IVC-based* system has been implemented recently [18].

DLP-based: KOE/IVC-based systems require a *private setup* phase that is discussed below. [48] (cf. [69]) assumes hardness of the Discrete Logarithm Problem (DLP) to build a system that requires only a *public setup*, like SCI.

Proof length in the initial works above was $\Theta\left(\sqrt{T}\right)$ and this was reduced to poly $\log T$ in [30], which also implemented both versions; verifier running time in both variants is $\Omega(T)$.

Comparing SCI *to prior CI solutions* SCI is the first CI solution that achieves both (1) a short public randomness setup phase and (2) universal scalability for one-shot computation. We discuss the significance of these properties after explaining them. (A quantitative comparison of the running time, memory consumption and communication complexity of SCI to prior systems appears in Section 2 and Table 1.)

One-shot universal scalability (OSUS) A CI system is *universally scalable* if for any fixed program $\mathbb{P}$, prover running time is bounded by Tpoly $\log T$ and verification time is at most poly $\log T$ where T is the number of machine cycles[9]. If the same asymptotic running times hold even for a single execution of $\mathbb{P}$, and where the setup ("preprocessing") is carried out by the verifier (and hence setup-cost is part of the total verification-cost), we shall say that CI solution is *one-shot universally scalable* (OSUS). DLP-based systems have super-linear verification time, hence are not *scalable* for any program. IP-based systems are efficient only for highly-parallel computations, thus are not *universally scalable*. LPCP- and KOE-based systems are universally scalable but not OSUS because they require a proving key k_P that is longer than T which must be generated by the verifier (in the one-shot setting). Of all prior solutions, only the IVC-based one is OSUS, like SCI.

Public setup All implemented solutions but for DLP-based and SCI, if instantiated as publicly verifiable CI systems, require a setup phase ("preprocessing"), the output of which is a pair of keys (k_P, k_V), one needed for proving statements, the other for verifying them. A "trapdoor key" k_{tpdr} is associated with (k_P, k_V) and can be used to forge pseudo-proofs of false statements. Furthermore, k_{tpdr} can be recovered by the parties that run the preprocessing phase. Secure multi-party computation can boost security by "distributing knowledge" of the trapdoor among several parties [17] so that all of them have to be compromised to recover k_{tpdr}; but this does not remove the concern that k_{tpdr} has been recovered by collusion of all parties, or retrieved by a central party eavesdropping to all of them. Even if k_{tpdr} has not been recovered by anyone, its mere existence may erode trust in such systems. (Cf. [6] for a recent discussion of setup-attacks and their implications and mitigations.) In contrast, SCI and DLP-based systems require only a short public random string when instantiated as a publicly verifiable noninteractive CI system.

Discussion The combination of OSUS and public setup which is unique to SCI has three implications: (i) the ease of setting up and modifying CI systems based on it is relatively small, (ii) the trust assumptions made by parties using it are comparatively minor and hence (iii) it seems more suitable than existing

---

[9] Formally, a CI system is *universally scalable* if for any language $L \in \textbf{NTIME}(T(n))$) prover running time is $T(n)$poly $\log T(n)$ and verifier running time is poly $\log T(n)$ where n denotes input length.

solutions for use in decentralized and public settings, like Bitcoin. We repeat and stress that many such applications require zero-knowledge proofs, a property achieved by prior solutions and not achieved by SCI; augmenting SCI to obtain zero knowledge seems within reach [11] but is outside the scope of our work.

SCI— *Main technical contributions* We faced three major challenges when attempting to construct PCP systems that scale well and apply to general programs, and SCI is the first implementation to contain scalable solutions to each of them, reported here for the first time: (i) implementing the recursive *proof composition* [2] technique applied to PCPs of proximity (PCPPs) [20, 36] (ii) constructing quasi-linear PCPP systems for Reed-Solomon (RS) error correcting codes [68] of huge message length [23] that require, in particular, quasi-linear time algorithms for interpolation and multi-point evaluation of large-degree polynomials over finite fields of characteristic 2; and (iii) reducing general programs that include jumps, loops, and random access memory (RAM) instructions to *succinct Algebraic Constraint Satisfaction Problem* (sACSP) instances that "capture" the corresponding CI statement (*); prior arithmetization solutions require the verifier, or a party trusted by it, to "unroll" a T-cycle computation to obtain an arithmetic circuit of size $\Omega(T)$, whereas SCI's verifier is succcint and does not perform this unrolling. (All prior solutions arithmetize over large prime fields; SCI is also novel in its being the first arithmetization over large binary fields, which poses new challenges, especially for integer operations like addition and multiplciation, cf. Section B.1.)

To overcome the blowup (i) that is due to recursive PCPP composition, we replace PCPPs with *interactive oracle proofs of proximity (IOPPs)* [38, 9, 10], implemented here for the first time, and increase the number of rounds of interaction between prover and verifier; the extra rounds can be removed in the random oracle model [38]. To address (ii) we built a dedicated library that implements finite field arithmetic efficiently (reported in [22]) and used it to further implement additive Fast Fourier Transforms (aFFT) [40] that perform interpolation and multi-point evaluation in quasi-linear time and in parallel (via multi-threading); the large-scale additive FFTs are reported here for the first time. To solve (iii) and reduce general programs to PCP systems efficiently, we devise a novel reduction from general programs for random access machines to sACSP instances. We describe these three contributions in more detail in Section 3 and the appendix.

2 Measurements

SCI can be applied to any language in **NEXP**; for concreteness we picked two programs computing the NP-complete subset-sum problem (cf. Appendix C); we explain this choice after introducing the two programs. The input to the subset-sum problem is an integer array A of size n and a target integer t; the problem is to decide whether there exists a subset $A' \subset A$ that sums to t. The CI statement addressed here is the co-NP version of the problem, stating "no

subset of A sums to t" and denoted by $\tau_{(A,n,t)}$. The two programs differ in their time and space consumption. The first one *exhaustively* tries all possible subsets, requiring 2^n cycles but only $O(1)$ memory, hence can be executed using only the local registers of the machine and with no random access to memory. The second program uses *sorting* and runs in time $O(2^{n/2})$, a quadratic improvement over the exhaustive solution but it also requires $\Theta(2^{n/2})$ memory and hence uses the random access memory. We denote the two programs by $\mathbb{P}_{\mathrm{exh}}$ and $\mathbb{P}_{\mathrm{sort}}$, respectively.

On choice of programs We would like to run SCI on "real-world" applications like the examples given in the introduction but our current scalability is not up to par. This situation is similar to that of the very first works on other CI solutions (cf. [34, 70, 66, 15]): initial reports discussed only small word-size machines, restricted functionality and simple programs. Like some of those works (most notably, [19]) we use the 16-bit version of the TinyRAM architecture as our model of computation, and support all of its assembly code even though these two programs use only a subset of it. We focus on subset-sum for two reasons: (i) it is a natural NP-complete problem that is often used in cryptographic applications but more importantly (ii) it allows us to display the effect of time–space tradeoffs on our CI solution (cf. Figure 2). Since SCI supports non-determinism, we could have used the non-deterministic version of the subset-sum statement. In fact, this would have reduced prover and verifier complexity because fewer boundary constraints are imposed on the input. However, the resulting statement seems less interesting, saying "there exists A such that no subset of it sums to t".

Measurement range Input array size n ranged between 3–16. Prover data was measured on a "large" server with 32 AMD Opteron cores at clock rate 3.2 GHz and 512 Gigabytes of RAM, running with two threads per core (total of 64 threads); to bound the single-core/thread prover time one may multiply the stated times by $\times 32/\times 64$ respectively. Verifier data was measured on a "standard" laptop, a Lenovo T440s with Intel core i7-4600 at clock rate 2.1 GHz and 12 Gigabytes RAM. We stress that verifier succinctness for one-shot programs allows us to measure verifier running time independently of prover running time, all the way up to 2^{47} machine cycles. Both prover and verifier were measured for 1-bit security and 80-bit security using state-of-the-art PCPP and IOPP security estimates [9].

Prover time and memory The left column of Figure 1 presents the running time (top) and memory consumption (bottom) of the Prover for both $\mathbb{P}_{\mathrm{exh}}$ and $\mathbb{P}_{\mathrm{sort}}$ as a function of the number of machine cycles of the simulated machine for both 1-bit and 80-bit security level. The two main observations from these figures are that (i) resources scale quasi-linearly with number of cycles and (ii) $\mathbb{P}_{\mathrm{sort}}$ is more costly than $\mathbb{P}_{\mathrm{exh}}$ due to its random access memory usage, which increases proof length by $\times \log^{O(1)} T$ factor for a T-cycle execution (cf. Section 3). Figure 2 compares time and memory as a function of the size on the input array n and shows that for $n \geq 8$ the quadratic running-time improvement of $\mathbb{P}_{\mathrm{sort}}$ over $\mathbb{P}_{\mathrm{exh}}$

outweighs the $\times O(\log T)$ factor required by random access to memory, both for 1-bit and 80-bit security level.

Verifier time and query complexity The right column of Figure 1 shows verifier running time (top) and query complexity (bottom) for both programs for both 1-bit and 80-bit security levels. Notice the $\approx 2^{13}$–$2^{23}\times$ factor improvement of verifier over prover in both parameters (recall $1MB = 2^{10}KB$) and the increase in running time as a function of security due to repetition. For small n verifier running time is greater than that of the naïve verifier which re-runs the program. However, since naive verification grows like 2^n for $\mathbb{P}_{\text{exh}}$ and like $2^{n/2}$ for $\mathbb{P}_{\text{sort}}$, for $n \geq 22$ (at 80-bit security) our verifier is more efficient than the naïve one for $\mathbb{P}_{\text{exh}}$, and for $n \geq 48$ the verifier for $\mathbb{P}_{\text{sort}}$ is more efficient than the naïve one (cf. Figure 3).

Table 1: Quantitative comparison of SCI with KOE-based [15], IVC-based [18] and DLP-based [48] solutions. Data measured on executions of 2^{16} cycles of $\mathbb{P}_{\text{exh}}$ at an 80-bit security level on the same machine with 32 AMD Opteron cores at clock rate 3.2 GHz and 512 Gigabytes of RAM. The DLP-based column is extrapolated from [48, Table 2], accounting for (i) the larger circuit size of our computation (which has ~ 132M gates compared with maximal size of 1.4M gates there) and different compute architectures (single threaded Intel 4690K core vs. 64 threaded AMD Opteron). Notice the proving time of SCI is $\sim \times 2 - 4$ slower than KOE- and DLP-based and $\sim \times 150$ faster than IVC-based. Regarding total communication complexity, SCI is more efficient than prior solutions but less efficient when measuring only post-processing communication.

		KOE-based	**IVC-based**	**DLP-based**	**SCI**
Verifier setup	time	~ 28 min	~ 10 sec	~ 0.7 sec	<0.01 sec
	key length	~ 18.9 GB	43 MB	154 MB	16 bytes
Prover	time	~ 18 min	4.2 days	~ 8 min	~ 41 min
	memory	~ 216 GB	2.9 GB	~ 1 TB	~ 135 GB
Verifier postprocessing	time	<10 ms	~ 25 ms	~ 1.7 min	~ 0.5 sec
	communication complexity	230 bytes	374 bytes	8.8KB	~ 42.5 MB
Verifer total	time	~ 28 min	~ 10 sec	1.7 min	~ 0.5 sec
	communication complexity	~ 18.9 GB	43 MB	~ 154 MB	~ 42.5 MB

Quantitative comparison with other CI implementations Table 1 compares SCI to three recent CI systems, the KOE-based [15], the IVC-based [18], and the DLP-based [48], using the version with $\text{poly} \log(T)$ communication complexity. One sees that SCI has the shortest and fastest setup but larger post-setup communication complexity; post-setup verification is faster than DLP-based but slower than KOE/IVC-based, as predicted by theory. Two other important points are: (i) proofs in SCI are not zero-knowledge whereas the other solutions are, and (ii) the setup of the last two columns (DLP-based and SCI) is comprised only of

a public random string, whereas KOE/IVC-based solutions require private setup and involve a trapdoor that can be used to forge proofs of false statements.
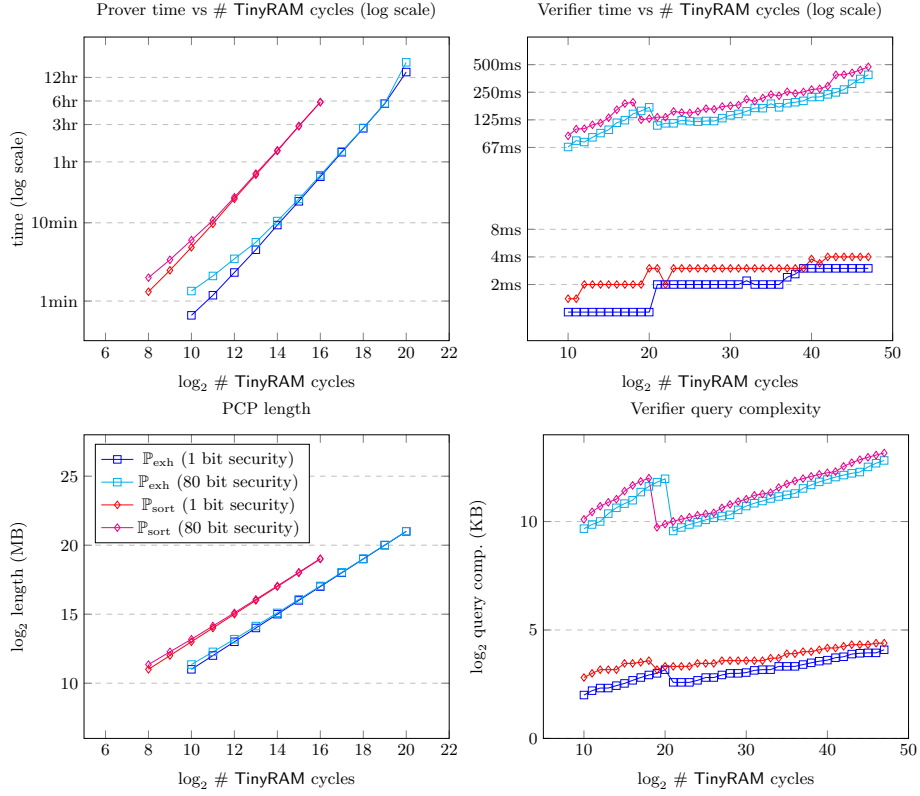

Fig. 1: Comparison of prover (left) and verifier (right) running time (top) and memory consumption (bottom). The sharp drop in query complexity is due to transition from 2 to 3 levels of recursion in the RS-PCPP; as seen in the top-right, this has little effect on overall verifier running time, which is significantly smaller than prover running time, and also grows at a considerably slower rate as a function of # cycles. Answers to verifier queries provided by random strings which simulates accurately actual proofs because verifier is non-adaptive, i.e., its running time is independent of the proof content.
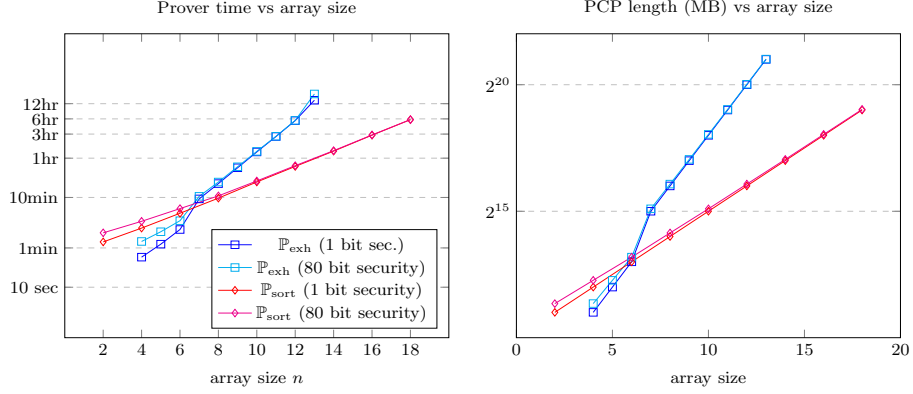
9

Fig. 2: Prover running time (left) and memory consumption (right) as a function of input array size n. For $n \geq 8$ the quadratic running-time improvement of $\mathbb{P}_{\text{sort}}$ over $\mathbb{P}_{\text{exh}}$ overcomes the $\times \text{poly} \log T$ factor overhead of $\mathbb{P}_{\text{exh}}$ due to random memory access; this holds for both 1-bit and 80-bit security level.
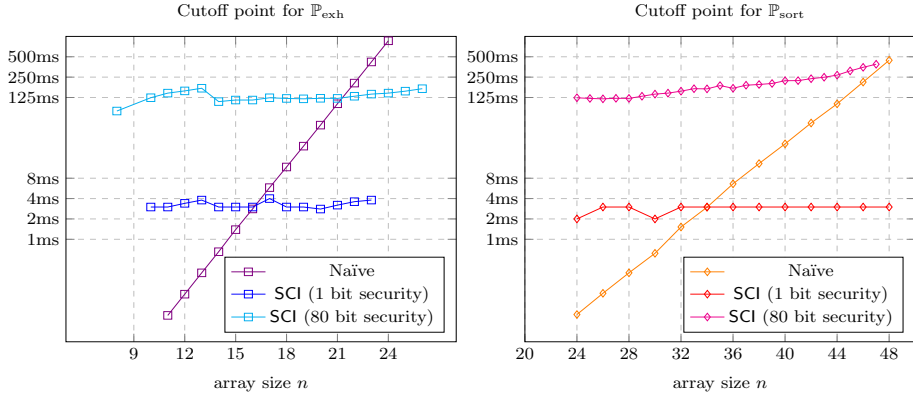

Fig. 3: Computation of the *break-even point* [73, 72], the minimal input size n for which naïve verification via re-execution becomes more costly than PCP-based verification. For $\mathbb{P}_{\text{exh}}$ at 80-bit security this threshold is at $n = 22$ and for $\mathbb{P}_{\text{sort}}$ it is significantly higher, estimated around $n = 48$, due to quadratic improvement in running time of the latter program.

3 Overview of construction

The construction of the PCP $\pi_{(\mathbb{P},x,y,T)}$ for the computational statement $\tau_{(\mathbb{P},x,y,T)}$ follows the rather complex process detailed in [23, 21, 14, 13] which we summarize next (see Appendix A).The statement $\tau_{(\mathbb{P},x,y,T)}$ is converted into an instance $\psi_{(\mathbb{P},x,y,T)}$ of an *algebraic constraint satisfaction problem* (ACSP) over a finite

field[10] $\mathbb{F}$ of characteristic 2 and $\tau_{(\mathbb{P},x,y,T)}$ is used by prover and verifier as described next.

Prover To construct the PCP, the prover executes $\mathbb{P}$ on input x and encodes the execution trace by a Reed-Solomon [68] codeword $\mathsf{a}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$ evaluated over an additive sub-group of $\mathbb{F}$. The ACSP instance $\psi_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$ is applied to $\mathsf{a}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$ as described in [23, Equation (3.2)] to obtain an additional RS-codeword, denoted $\mathsf{b}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})} = \psi_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}(\mathsf{a}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})})$, that "attests" to the fact that $\mathsf{a}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$ encodes a valid execution trace, and hence, in particular, its output is correct. Each of the two codewords is appended with a PCP of proximity (PCPP) for the RS-code [23], denoted $\pi_\mathsf{a}, \pi_\mathsf{b}$, respectively. The PCP $\pi_{(\mathbb{P},x,y,T)}$ is defined to be the concatenation of $\mathsf{a}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}, \mathsf{b}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}, \pi_\mathsf{a}$ and π_b.

Verifier The verifier queries the four parts of the PCP in the following manner: First it invokes an RS-PCPP verifier that queries $\mathsf{a}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$ and π_a to "check" that $\mathsf{a}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$ is close in Hamming distance to a codeword of the RS-code; it repeats this process with respect to $\mathsf{b}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$ and π_b. Second and last, the verifier queries $\mathsf{a}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$ and $\mathsf{b}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$ and uses $\psi_{(\mathbb{P},x,y,T)}$ to check that the two codewords encode a valid computation of $\mathbb{P}$ that starts with x and reaches y within T cycles. In this process we rely on the "locality" of the mapping $\psi_{(\mathbb{P},x,y,T)} : \mathsf{a}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})} \to \mathsf{b}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$ which means that each entry of $\mathsf{b}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$ depends on a small number of entries of $\mathsf{a}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$. In what follows we elaborate on the novel aspects of this reduction as implemented in SCI.

From assembly code to succinct ACSP The *efficiency* of the ACSP instance $\psi_{(\mathbb{P},x,y,T)}$ is measured by three parameters that we seek to minimize: *circuit size, degree,* and *query complexity,* denoted $C_{(\mathbb{P},x,y,T)}, D_{(\mathbb{P},x,y,T)}, Q_{(\mathbb{P},x,y,T)}$ respectively. Circuit size affects both proving and verification time; degree affects PCP length and reducing it decreases running time and memory consumption on the prover side; query complexity affects the length of communication between prover and verifier (and the length of computationally sound (CS) proofs $\hat{\pi}$) as well as verifier running time. Each parameter can be optimized at the expense of the other two, and the challenge is to reach an efficient balance between all three.

Our starting point is a program $\mathbb{P}$, i.e., a sequence of *instructions* for a random access machine (RAM). For simplicity we first focus on instructions that access only (local) registers; random access memory instructions are discussed below. Each instruction specifies the input and output register locations and an operation applied to the inputs, called the *opcode*. We build $\psi_{(\mathbb{P},x,y,T)}$ bottom-up (cf. Appendix B for a detailed example). Each opcode op appearing in $\mathbb{P}$ (like xor, add, jump, etc.) is specified by an *algebraic definition* over $\mathbb{F}$; in other words, we specify a set of multi-variate polynomials $\mathcal{P}_\mathsf{op} \subseteq \mathbb{F}[X_1, X_2, \ldots, X_m]$ such that the set of common zeros of $\mathcal{P}_\mathsf{op}$ correspond to correct input-output tuples for op. Program flow is controlled by multiplying each polynomial in $\mathcal{P}_\mathsf{op}$ by a multivariate Lagrange "selector" polynomial that, based on the value v of the

---

[10] SCI uses the field of size 2^{64} which suffices for the computations measured here.

program counter (PC), annihilates all constraints that are irrelevant for enforcing the vth instruction of $\mathbb{P}$. For a program with ℓ lines these selector polynomials have degree $\lceil \log \ell \rceil$. The resulting ACSP has circuit size $O(\ell)$ and degree and query complexity are $\log \ell + O(1)$; the constants hidden by asymptotic notation depend on the machine specification.

Random access memory instructions The *execution trace* of $\mathbb{P}$ is the length–T sequence of machine states that describes the computation. To verify the integrity of random access memory instructions (such as load and store) we follow [13, 14] and use a *pair* of execution traces. The first trace, trace$^{\text{time}}$, is sorted increasingly by time, and the second, trace$^{\text{mem}}$, is sorted lexicographically first by memory location, then by time. RAM-related execution validity is verified "locally" by inspecting pairs of consecutive elements in trace$^{\text{mem}}$, just like non-RAM related instructions are verified "locally" by inspecting pairs of consecutive elements in trace$^{\text{time}}$. To further reduce proof length and query complexity, each state of trace$^{\text{mem}}$ contains only the information needed to check memory consistency — an address, its content and the type of memory access (load/store); let s denote the number of field elements in a single line of trace$^{\text{mem}}$.

To prove that trace$^{\text{mem}}$ and trace$^{\text{time}}$ refer to the same execution, the prover must describe a permutation between the two, and the verifier must check its validity. To achieve this SCI uses a non-blocking Beneš switching network [32, 25] embedded in an affine graph over $\mathbb{F}$ (cf. [23, 14] for definitions). Using this method, adding RAM-related instructions to a program adds only $O(T \cdot \log T)$ field elements to the PCP and increases query complexity by a small constant.

Reducing proof construction time via Interactive Oracle Proofs of Proximity (IOPP) A significant portion of the prover running time and memory consumption are dedicated to the construction of the PCP of Proximity (PCPP) for $\mathsf{a}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$ and for $\mathsf{b}_{(\mathbb{P},\mathsf{x},\mathsf{y},\mathsf{T})}$. The full PCPP for an RS-codeword of degree N is of length $O(N \log^{2.6} N)$ which is quite large in our applications. Observing that (i) these PCPPs are built using recursive *PCPP composition* [21], and (ii) only a small fraction of recursive branches are explored by the verifier, we increase the number of rounds of interaction and use a *notarized interactive proof of proximity* (NIPP) [9], a special case of *interactive oracle proofs of proximity* (IOPP) [38, 10] to reduce proof length to $4N + O(\sqrt{N})$. The added rounds of interaction can be removed in the random oracle model to obtain computationally sound proofs [38].

Parallel implementation of PCPPs for RS codes To reduce the time required to encode the execution trace into a pair of RS-codewords, SCI uses parallel algorithms for finite field operations and for dealing with polynomials over finite fields of characteristic 2. To speed up basic field operations (most notably, multiplication) a dedicated algebraic library was built, that utilizes parallel hardware on multi-core CPU. Interpolation and evaluation of polynomials over affine spaces of size N are computed in quasilinear time using so-called *additive Fast Fourier Transform* (aFFT) [40].

4 Concluding remarks

SCI is the first implementation of a system of computational integrity that achieves asymptotic one shot universal scalability (OSUS) with a setup key that is merely a public random string. Prior solutions either required super-linear verification time, or used a setup procedure that involves keys which could be used to forge proofs of falsities. While the computer programs on which SCI was tested are of limited applicability, the simpler setup assumptions of SCI make it a natural starting point for building further applications — most notably zero knowledge proofs — for use in decentralized networks.

References

1. Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification and the hardness of approximation problems. Journal of the ACM 45(3), 501–555 (1998), preliminary version in FOCS '92.
2. Arora, S., Safra, S.: Probabilistic checking of proofs: a new characterization of NP. Journal of the ACM 45(1), 70–122 (1998), preliminary version in FOCS '92.
3. Babai, L., Fortnow, L., Levin, L.A., Szegedy, M.: Checking computations in polylogarithmic time. In: Proceedings of the 23rd Annual ACM Symposium on Theory of Computing. pp. 21–32. STOC '91 (1991)
4. Babai, L., Fortnow, L., Lund, C.: Nondeterministic exponential time has two-prover interactive protocols. In: Proceedings of the 31st Annual Symposium on Foundations of Computer Science. pp. 16–25. SFCS '90 (1990)
5. Babai, L., Moran, S.: Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity class. Journal of Computer and System Sciences 36(2), 254–276 (1988)
6. Bellare, M., Fuchsbauer, G., Scafuro, A.: Nizks with an untrusted crs: Security in the face of parameter subversion. Cryptology ePrint Archive, Report 2016/372 (2016), http://eprint.iacr.org/
7. Bellare, M., Goldreich, O.: On defining proofs of knowledge. In: Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology. pp. 390–420. CRYPTO '92 (1993)

8. Ben-Or, M., Goldwasser, S., Kilian, J., Wigderson, A.: Multi-prover interactive proofs: how to remove intractability assumptions. In: Proceedings of the 20th Annual ACM Symposium on Theory of Computing. pp. 113–131. STOC '88 (1988)

9. Ben-Sasson, E., Ben-Tov, I., Gabizon, A., Riabzev, M.: Improved concrete efficiency and security analysis of reed-solomon pcpps (2016), `http://eccc.hpi-web.de/report/2016/073`

10. Ben-Sasson, E., Chiesa, A., Gabizon, A., Riabzev, M., Spooner, N.: Short interactive oracle proofs with constant query complexity, via composition and sumcheck. Electronic Colloquium on Computational Complexity (2016), tR16–046

11. Ben-Sasson, E., Chiesa, A., Gabizon, A., Virza, M.: Quasi-linear size zero knowledge from linear-algebraic PCPs. In: Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II. pp. 33–64 (2016)

12. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from Bitcoin. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy. pp. 459–474. SP '14 (2014)

13. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In: Proceedings of the 4th Innovations in Theoretical Computer Science Conference. pp. 401–414. ITCS '13 (2013)

14. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: On the concrete efficiency of probabilistically-checkable proofs. In: Proceedings of the 45th ACM Symposium on the Theory of Computing. pp. 585–594. STOC '13 (2013)

15. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: Verifying program executions succinctly and in zero knowledge. In: Proceedings of the 33rd Annual International Cryptology Conference. pp. 90–108. CRYPTO '13 (2013)

16. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: TinyRAM architecture specification v2.00 (2013), `http://scipr-lab.org/tinyram`

17. Ben-Sasson, E., Chiesa, A., Green, M., Tromer, E., Virza, M.: Secure sampling of public parameters for succinct zero knowledge proofs. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. pp. 287–304 (2015), `http://dx.doi.org/10.1109/SP.2015.25`

18. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. In: Proceedings of the 34th Annual International Cryptology Conference. pp. 276–294. CRYPTO '14 (2014)

19. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von Neumann architecture. In: Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014. pp. 781–796 (2014)

20. Ben-Sasson, E., Goldreich, O., Harsha, P., Sudan, M., Vadhan, S.: Short PCPs verifiable in polylogarithmic time. In: Proceedings of the 20th Annual IEEE Conference on Computational Complexity. pp. 120–134. CCC '05 (2005)

21. Ben-Sasson, E., Goldreich, O., Harsha, P., Sudan, M., Vadhan, S.: Robust PCPs of proximity, shorter PCPs, and applications to coding. SIAM Journal on Computing 36(4), 889–974 (2006), preliminary versions of this paper have appeared in Proceedings of the 36th ACM Symposium on Theory of Computing and in Electronic Colloquium on Computational Complexity.

22. Ben-Sasson, E., Hamilis, M., Silberstein, M., Tromer, E.: Fast multiplication in binary fields on gpus via register cache. In: Proceedings of the 2016 International Conference on Supercomputing. ICS '16 (2016)

23. Ben-Sasson, E., Sudan, M.: Short PCPs with polylog query complexity. SIAM Journal on Computing 38(2), 551–607 (2008), preliminary version appeared in STOC '05.
24. Ben-Sasson, E., Sudan, M., Vadhan, S., Wigderson, A.: Randomness-efficient low degree tests and short PCPs via epsilon-biased sets. In: Proceedings of the 35th Annual ACM Symposium on Theory of Computing. pp. 612–621. STOC '03 (2003)
25. Beneš, V.E.: Mathematical theory of connecting networks and telephone traffic (1965), `http://opac.inria.fr/record=b1083990`
26. Beneš, V.E.: Mathematical theory of connecting networks and telephone traffic. New York, Academic Press (1965)
27. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. pp. 326–349. ITCS '12 (2012)
28. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: Recursive composition and bootstrapping for SNARKs and proof-carrying data. In: Proceedings of the 45th ACM Symposium on the Theory of Computing. pp. 111–120. STOC '13 (2013)
29. Bitansky, N., Chiesa, A., Ishai, Y., Ostrovsky, R., Paneth, O.: Succinct non-interactive arguments via linear interactive proofs. In: Proceedings of the 10th Theory of Cryptography Conference. pp. 315–333. TCC '13 (2013)
30. Bootle, J., Cerulli, A., Chaidos, P., Groth, J., Petit, C.: Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In: Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II. pp. 327–357 (2016), `http://dx.doi.org/10.1007/978-3-662-49896-5_12`
31. Chiesa, A., Zhu, Z.A.: Shorter arithmetization of nondeterministic computations. Theoretical Computer Science 600, 107 – 131 (2015), `http://www.sciencedirect.com/science/article/pii/S0304397515006647`
32. Clos, C.: A study of non-blocking switching networks. Bell System Technical Journal 32(2), 406–424 (mar 1953), `http://dx.doi.org/10.1002/j.1538-7305.1953.tb01433.x`
33. Cormode, G., Mitzenmacher, M., Thaler, J.: Practical verified computation with streaming interactive proofs. In: Proceedings of the 4th Symposium on Innovations in Theoretical Computer Science. pp. 90–112. ITCS '12 (2012)
34. Cormode, G., Thaler, J., Yi, K.: Verifying computations with streaming interactive proofs. Proceedings of the VLDB Endowment 5(1), 25–36 (2011)
35. Dinur, I.: The PCP theorem by gap amplification. Journal of the ACM 54(3), 12 (2007)
36. Dinur, I., Reingold, O.: Assignment testers: Towards a combinatorial proof of the PCP theorem. SIAM J. Comput. 36(4), 975–1024 (2006), `http://dx.doi.org/10.1137/S0097539705446962`
37. Dwork, C., Feige, U., Kilian, J., Naor, M., Safra, S.: Low communication 2-prover zero-knowledge proofs for NP. In: Proceedings of the 12th Annual International Cryptology Conference. pp. 215–227. CRYPTO '92 (1992)
38. Eli Ben-Sasson, Alessandro Chiesa, N.S.: Interactive oracle proofs. IACR Cryptology ePrint Archive 2016, 116 (2016), `http://eprint.iacr.org/2016/116`
39. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Proceedings of the 6th Annual International Cryptology Conference. pp. 186–194. CRYPTO '87 (1987)

40. Gao, S., Mateer, T.: Additive fast fourier transforms over finite fields. IEEE Trans. Inf. Theor. 56(12), 6265–6272 (Dec 2010), `http://dx.doi.org/10.1109/TIT.2010.2079016`

41. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: Proceedings of the 30th Annual Conference on Advances in Cryptology. pp. 465–482. CRYPTO'10, Springer-Verlag, Berlin, Heidelberg (2010), `http://dl.acm.org/citation.cfm?id=1881412.1881445`

42. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: Proceedings of the 32nd Annual International Conference on Theory and Application of Cryptographic Techniques. pp. 626–645. EUROCRYPT '13 (2013)

43. Goldreich, O., Sudan, M.: Locally testable codes and PCPs of almost-linear length. Journal of the ACM 53, 558–655 (July 2006), preliminary version in STOC '02.

44. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: Interactive proofs for Muggles. In: Proceedings of the 40th Annual ACM Symposium on Theory of Computing. pp. 113–122. STOC '08 (2008)

45. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM Journal on Computing 18(1), 186–208 (1989), preliminary version appeared in STOC '85.

46. Greenberg, A.: Zcash, an untraceable bitcoin alternative, launches in alpha (January 2016), [Wired.com; posted online 20-January-2016]

47. Groth, J.: Short pairing-based non-interactive zero-knowledge arguments. In: Proceedings of the 16th International Conference on the Theory and Application of Cryptology and Information Security. pp. 321–340. ASIACRYPT '10 (2010)

48. Groth, J.: Efficient zero-knowledge arguments from two-tiered homomorphic commitments. In: Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings. pp. 431–448 (2011), `http://dx.doi.org/10.1007/978-3-642-25385-0_23`

49. Harsha, P., Sudan, M.: Small PCPs with low query complexity. Computational Complexity 9(3–4), 157–201 (Dec 2000), preliminary version in STACS '91.

50. Håstad, J.: Some optimal inapproximability results. Journal of the ACM 48(4), 798–859 (2001)

51. Horowitz, E., Sahni, S.: Computing partitions with applications to the knapsack problem. J. ACM 21(2), 277–292 (1974), `http://doi.acm.org/10.1145/321812.321823`

52. Ishai, Y., Kushilevitz, E., Ostrovsky, R.: Efficient arguments without short PCPs. In: Proceedings of the Twenty-Second Annual IEEE Conference on Computational Complexity. pp. 278–291. CCC '07 (2007)

53. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge proofs from secure multiparty computation. SIAM J. Comput. 39(3), 1121–1152 (2009)

54. Ishai, Y., Mahmoody, M., Sahai, A.: On efficient zero-knowledge PCPs. In: Proceedings of the 9th Theory of Cryptography Conference on Theory of Cryptography. pp. 151–168. TCC '12 (2012)

55. Ishai, Y., Mahmoody, M., Sahai, A., Xiao, D.: On zero-knowledge PCPs: Limitations, simplifications, and applications (2015), `http://www.cs.virginia.edu/~mohammad/files/papers/ZKPCPs-Full.pdf`, available online

56. Kilian, J.: A note on efficient zero-knowledge proofs and arguments. In: Proceedings of the 24th Annual ACM Symposium on Theory of Computing. pp. 723–732. STOC '92 (1992)

57. Kilian, J., Petrank, E., Tardos, G.: Probabilistically checkable proofs with zero knowledge. In: Proceedings of the 29th Annual ACM Symposium on Theory of Computing. pp. 496–505. STOC '97 (1997)
58. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. Cryptology ePrint Archive, Report 2015/675 (2015), `http://eprint.iacr.org/`
59. Lipmaa, H.: Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In: Proceedings of the 9th Theory of Cryptography Conference on Theory of Cryptography. pp. 169–189. TCC '12 (2012)
60. Lund, C., Fortnow, L., Karloff, H., Nisan, N.: Algebraic methods for interactive proof systems. J. ACM 39(4), 859–868 (Oct 1992), `http://doi.acm.org/10.1145/146585.146605`
61. Mahmoody, M., Xiao, D.: Languages with efficient zero-knowledge pcps are in SZK. In: Proceedings of the 10th Theory of Cryptography Conference. pp. 297–314. TCC '13 (2013)
62. Micali, S.: Computationally sound proofs. SIAM Journal on Computing 30(4), 1253–1298 (2000), preliminary version appeared in FOCS '94.
63. Mie, T.: Short PCPPs verifiable in polylogarithmic time with o(1) queries. Annals of Mathematics and Artificial Intelligence 56, 313–338 (2009)
64. Moshkovitz, D., Raz, R.: Two-query PCP with subconstant error. Journal of the ACM 57, 1–29 (June 2008), preliminary version appeared in FOCS '08.
65. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (May 2009), `http://www.bitcoin.org/bitcoin.pdf`
66. Parno, B., Gentry, C., Howell, J., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: Proceedings of the 34th IEEE Symposium on Security and Privacy. pp. 238–252. Oakland '13 (2013)
67. Raz, R.: A parallel repetition theorem. In: Proceedings of the 27th Annual ACM Symposium on Theory of Computing. pp. 447–456. STOC '95 (1995)
68. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. Journal of the Society for Industrial and Applied Mathematics 8(2), 300–304 (1960), `http://dx.doi.org/10.1137/0108018`
69. Seo, J.H.: Round-efficient sub-linear zero-knowledge arguments for linear algebra. In: Public Key Cryptography - PKC 2011 - 14th International Conference on Practice and Theory in Public Key Cryptography, Taormina, Italy, March 6-9, 2011. Proceedings. pp. 387–402 (2011), `http://dx.doi.org/10.1007/978-3-642-19379-8_24`
70. Setty, S., Blumberg, A.J., Walfish, M.: Toward practical and unconditional verification of remote computations. In: Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems. pp. 29–29. HotOS '11 (2011)
71. Setty, S., Braun, B., Vu, V., Blumberg, A.J., Parno, B., Walfish, M.: Resolving the conflict between generality and plausibility in verified computation. In: Proceedings of the 8th EuoroSys Conference. pp. 71–84. EuroSys '13 (2013)
72. Setty, S., McPherson, M., Blumberg, A.J., Walfish, M.: Making argument systems for outsourced computation practical (sometimes). In: Proceedings of the 2012 Network and Distributed System Security Symposium. NDSS '12 (2012)
73. Setty, S., Vu, V., Panpalia, N., Braun, B., Blumberg, A.J., Walfish, M.: Taking proof-based verified computation a few steps closer to practicality. In: Proceedings of the 21st USENIX Security Symposium. pp. 253–268. Security '12 (2012)
74. Thaler, J.: Time-optimal interactive proofs for circuit evaluation. In: Proceedings of the 33rd Annual International Cryptology Conference. pp. 71–89. CRYPTO '13 (2013)

75. Valiant, P.: Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In: Proceedings of the 5th Theory of Cryptography Conference. pp. 1–18. TCC '08 (2008)
76. Vu, V., Setty, S., Blumberg, A.J., Walfish, M.: A hybrid architecture for interactive verifiable computation. In: Proceedings of the 34th IEEE Symposium on Security and Privacy. pp. 223–237. Oakland '13 (2013)
77. Wahby, R.S., Setty, S.T.V., Ren, Z., Blumberg, A.J., Walfish, M.: Efficient RAM and control flow in verifiable outsourced computation. In: 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014 (2015)
78. Walfish, M., Blumberg, A.J.: Verifying computations without reexecuting them. Commun. ACM 58(2), 74–84 (2015), http://doi.acm.org/10.1145/2641562

A Detailed PCP construction

We describe the way a PCP is generated for $\tau_{(\mathbb{P},x,y,T)}$, then discuss its verification.

Proof generation The PCP proof $\pi_{(\mathbb{P},x,y,T)}$ for $\tau_{(\mathbb{P},x,y,T)}$ is a concatenation of four sub-proofs: two codewords in a Reed-Solomon code [68] and two quasilinear size PCPs of Proximity (PCPP) for the RS-codewords [23]. To obtain these four sub-proofs, the prover starts by executing the program $\mathbb{P}$ on input x for T steps and records its *execution trace* — the length–T sequence of machine states that the machine goes through during execution. Each state is converted to a sequence of elements in the finite field $\mathbb{F}$ of size 2^{64}; Auxiliary field elements are appended to each state to reduce the degree complexity of $\psi_{(\mathbb{P},x,y,T)}$ as described in Section B; let s denote the total number of field elements per state. The resulting *algebraic trace* trace^{aug} is thus a table of $N = T \cdot s$ elements of $\mathbb{F}$, and is viewed as a function from $S \subset \mathbb{F}, |S| = N$ to $\mathbb{F}$, where S is an affine space over the two-element field. Prover now computes the *low-degree extension* (LDE) of trace^{aug} by interpolating and then evaluating trace^{aug} on a set $S' \subset \mathbb{F}$ that is significantly larger than S. This results in a codeword $\mathsf{a}_{(\mathbb{P},x,y,T)}$ of a Reed-Solomon (RS) code [68] over $\mathbb{F}$ of degree $N - 1$ and rate $\rho = |S|/|S'|$. Next, the ACSP instance $\psi_{(\mathbb{P},x,y,T)}$ is applied to $\mathsf{a}_{(\mathbb{P},x,y,T)}$ as described in [23, Equation (3.2)], producing another RS-codeword $\mathsf{b}_{(\mathbb{P},x,y,T)} = \psi_{(\mathbb{P},x,y,T)}(\mathsf{a}_{(\mathbb{P},x,y,T)})$, of degree $D_{(\mathbb{P},x,y,T)} \cdot (N - 1)$ and rate $\rho' = D_{(\mathbb{P},x,y,T)} \cdot \rho$ (SCI uses $\rho' = \frac{1}{8}$). Finally, a PCP of proximity (PCPP) for RS-codes [23] is appended to each of $\mathsf{a}_{(\mathbb{P},x,y,T)}$ and $\mathsf{b}_{(\mathbb{P},x,y,T)}$ to prove that indeed each belongs to the RS-code of the designated rate — ρ for $\mathsf{a}_{(\mathbb{P},x,y,T)}$ and ρ' for $\mathsf{b}_{(\mathbb{P},x,y,T)}$; denote these PCPPs by $\pi_{\mathsf{a}}, \pi_{\mathsf{b}}$, respectively. Summing up, the PCP proof $\pi_{(\mathbb{P},x,y,T)}$ is the concatenation of the four strings $\mathsf{a}_{(\mathbb{P},x,y,T)}, \pi_{\mathsf{a}}, \mathsf{b}_{(\mathbb{P},x,y,T)}$ and π_{b}.

Proof verification On the verifier side, given $\psi_{(\mathbb{P},x,y,T)}$ as input and oracle access to
$$\pi_{(\mathbb{P},x,y,T)} = (\mathsf{a}_{(\mathbb{P},x,y,T)}, \pi_{\mathsf{a}}, \mathsf{b}_{(\mathbb{P},x,y,T)}, \pi_{\mathsf{b}})$$
as above, the verifier invokes the RS-PCPP verifier of [23] on each of $(\mathsf{a}_{(\mathbb{P},x,y,T)}, \pi_{\mathsf{a}})$ and $(\mathsf{b}_{(\mathbb{P},x,y,T)}, \pi_{\mathsf{b}})$. Then it checks that $\mathsf{a}_{(\mathbb{P},x,y,T)} = \psi_{(\mathbb{P},x,y,T)}(\mathsf{b}_{(\mathbb{P},x,y,T)})$ by sampling

both $a_{(\mathbb{P},x,y,T)}$ and $b_{(\mathbb{P},x,y,T)}$ at a small number of locations $(1 + Q_{(\mathbb{P},x,y,T)}$ per test). To boost soundness, each of the aforementioned tests is repeated a number of times, using fresh randomness (SCI uses 14 repetitions to reduce the probability of error to $\mathsf{error} = \frac{1}{2}$). The verifier "accepts" $\tau_{(\mathbb{P},x,y,T)}$ (i.e., proclaims it to be likely true) if and only if $\pi_{(\mathbb{P},x,y,T)}$ passes all these checks; the security analysis guarantees that this verdict is correct with probability $1 - \mathsf{error}$.

B   Algebraic definition of general programs as zero locus of low-degree polynomial system

Our goal here is to explain how SCI converts programs into succinct algebraic CSP (ACSP) instances. For concreteness this is described for the TinyRAM machine specification [16] — a simple random access machine (RAM) with 16 registers and 16-bit size words that includes opcodes for logical operations, integer arithmetic, conditional jumps and random access memory instructions; the same techniques could be adapted to other machine specifications.

Algebra preliminaries Fix a basis $\beta_0, \ldots, \beta_{63}$ for $\mathbb{F}_{2^{64}}$ over $\mathbb{F}_2$ generated by an irreducible polynomial $h(X)$. Any sequence of w bits $a_0, \ldots, a_{w-1}$ can be naturally mapped to the field element $\sum_{i=0}^{w-1} a_i \beta_i$ as long as $w < 64$ and vice versa, field elements can be converted to sequences of bits; we assume this natural mapping and in particular will often identify the a 16-bit sequence $(a_0, \ldots, a_{15})$ with the field element $\sum_{i=0}^{15} a_i \beta_i$.

Overview of reduction The reduction from RAM programs to ACSPs has been described in detail in [13] and further improved in [31]; we follow this route. In particular, instructions that involve the random access memory are verified using affine routing networks as explained in [13] (cf. [31]), although SCI uses an affine graph in which the Beneš network [26] is embedded. Boundary constraints (such as the initial and final state of the machine) are enforced as explained in [13]. A remaining problem of great practical importance that remained from previous works has been how to reduce efficiently the transition function described by a program into a set of low-degree polynomials whose zero-locus corresponds to a valid evolution of the program's transition function. We describe this below. Our reduction works bottom up and has two main steps. (i) First, we define the input–output relation of each opcode as the zero-locus of a system of low-degree polynomials. (ii) In similar manner we define the transition function of the program as the zero-locus of a (larger) system of polynomials, one that uses the definitions of opcodes in terms of polynomials. The resulting set of polynomials is "glued" into a single large polynomial as described, e.g., in [23, Equation (5.5)] and [13, Section 10].

B.1   Algebraic definition of opcodes

Our basic data-unit is called a *word*, in TinyRAM its size is 16 bits. The atoms of a computer program are *opcodes*; each opcode has a fixed amount of input

and output words. For example, XOR receives two words $A = (a_0, \ldots, a_{15})$, $B = (b_0, \ldots, b_{15})$ and its output is a single word $C = (c_0, \ldots, c_{15})$ where $c_i = a_i \oplus b_i$ and $\oplus$ denotes exclusive-or; the AND opcode outputs $c_i = a_i \wedge b_i$, the ADD opcode performs integer addition, etc. (cf. [16] for details).

An opcode op with k inputs and ℓ outputs defines a *relation* R_{op} that contains all sequences of inputs and outputs that correspond to valid executions of op. Continuing with the examples above and using f to denote the flag,

$$R_{\mathsf{XOR}} = \left\{ (a, b, c) \in \{0, 1\}^{3 \cdot 16} \mid a_i \oplus b_i \oplus c_i = 0 \right\}$$

$$R_{\mathsf{AND}} = \left\{ (a, b, c) \in \{0, 1\}^{3 \cdot 16} \mid (a_i \wedge b_i) \oplus c_i = 0 \right\}$$

$$R_{\mathsf{ADD}} = \left\{ (a, b, c) \in \{0, 1\}^{3 \cdot 16}, f \in \{0, 1\} \mid \sum_{i=0}^{15} a_i 2^i + \sum_{i=0}^{15} b_i 2^i - \left(f \cdot 2^{16} + \sum_{i=0}^{15} c_i 2^i \right) = 0 \right\}$$

An *algebraic opcode* is an opcode (as defined above) over an alphabet that is a finite field, i.e., $R_{\mathsf{op}} \subset \mathbb{F}^{k+\ell}$. Any finite set is an *algebraic set*, meaning it can be described as the zero-locus of a system of polynomials, however, these polynomials may have large degree and/or large arithmetic complexity, which would harm the efficiency of our reduction. To reduce degree and arithmetic complexity we shall allow *auxiliary* variables and consider algebraic sets S over $\mathbb{F}^{k+\ell+m}$ such that R_{op} is the projection of S to the first $k + \ell$ variables. Formally, an *algebraic constraint system* A_{op} corresponding to an opcode op with k inputs and ℓ outputs is a set of polynomials $A_{\mathsf{op}} \subset \mathbb{F}[X_1, \ldots, X_k, Y_1, \ldots, Y_\ell, Z_1, \ldots, Z_m]$ such that

$$R_{\mathsf{op}} = \{x_1, \ldots, x_k, y_1, \ldots, y_\ell \mid \exists z_1, \ldots, z_m, A_{\mathsf{op}}(x_1, \ldots, x_k, y_1, \ldots, y_\ell, z_1, \ldots, z_m) = 0\} \tag{1}$$

We call $X_1, \ldots, X_k$ the *input* variables, $Y_1, \ldots, Y_\ell$ the *output* variables and $Z_1, \ldots, Z_m$ are *auxiliary* variables. While any relation can be defined without any auxiliary variables, the degree of such A_{op} may be very large (e.g., in the case of AND, ADD), therefor, to minimize ACSP degree we shall often use auxiliary variables as shown in the following examples; explanations appear below but notice XOR uses no auxiliary variables and the AND opcode uses 48 of them. We defer the explanation of the more complicated ADD opcode to later on.

$$A_{\mathsf{XOR}} = \{X_1 + X_2 + Y_1\} \tag{2}$$

$$A_{\mathsf{AND}} = \left\{ X_1 + \sum_{i=0}^{15} Z_i \beta_i, X_2 + \sum_{i=0}^{15} Z_{16+i} \beta_i, Y_1 + \sum_{i=0}^{15} Z_{32+i} \beta_i \right\} \tag{3}$$

$$\bigcup \{Z_j \cdot (Z_j + 1) \mid j = 0, \ldots, 47\} \tag{4}$$

$$\bigcup \{(Z_i \cdot Z_{16+i}) + Z_{32+i} \mid i = 0, \ldots, 15\} \tag{5}$$

Recall that addition in $\mathbb{F}$ corresponds to exclusive-or, hence XOR has an algebraic constraint system with a single polynomial of degree 1 and no auxiliary

variables, and it satisfies (1). To see that (3)–(5) form an algebraic constraint system for AND we argue as follows. Suppose $(x_1, x_2, y_1, z_0, \ldots, z_{47})$ belongs to the zero-locus of A_{AND}, i.e., all polynomials in A_{AND} vanish on this input. Then by (4) we have $z_j \in \{0, 1\}$ for $j = 0, \ldots, 47$. By (3) we see that $z_{32+i} = z_i \wedge z_{16+i}$ for $i = 0, \ldots, 15$. Finally, by (3) we see that x_1 "packs" $z_0, \ldots, z_{15}$ into a single field elements, meaning x_1 is the field element whose representation in the basis $\beta_0, \ldots, \beta_{63}$ is the sequence $z_0, \ldots, z_{15}, 0, 0, \ldots, 0$ and similarly x_2 "packs" $z_{16}, \ldots, z_{31}$ and y_1 "packs" $z_{32}, \ldots, z_{47}$. Therefore, y_1 is the bitwise and of x_1 and x_2, as required by (1).

The constraints of the ADD opcode correspond to the operation of a full binary adder and appear below ((6)– (10)). In what follows auxiliary variables $Z_0, \ldots, Z_{15}$ are used to "unpack" X_1, auxiliary variables $Z_{16}, \ldots, Z_{31}$ "unpack" X_2, auxiliary variables $Z_{32}, \ldots, Z_{47}$ are the carry bits and $Z_{48}, \ldots, Z_{63}$ "unpack" the output Y_1; the overflow flag is stored in Y_2. The constraint set (6) "unpacks" both inputs and the output using 16 auxiliary variables each as done in (3) above. The constraint set (7) checks that each auxiliary variable is boolean (as done in (4)) but now we have 16 additional auxiliary variables for the carry bits, reaching a total of 64 auxiliary variables. The set of constraints (8) checks that the carry bits $(Z_{32}, \ldots, Z_{47})$ are computed correctly. In (9) the output is checked to be equal to the exclusive-or of the relevant input and carry bits. Finally, in (10) we check that the least significant carry and output bits are correct, and that the most significant carry bit (Z_{47}) equals the overflow flag (Y_2).

$$A_{\mathsf{ADD}} = \left\{ X_1 + \sum_{i=0}^{15} Z_i \beta_i, X_2 + \sum_{i=0}^{15} Z_{16+i} \beta_i, Y_1 + \sum_{i=0}^{15} Z_{48+i} \beta_i \right\} \tag{6}$$

$$\bigcup \{ Z_j \cdot (Z_i + 1) \mid j = 0, \ldots, 63 \} \tag{7}$$

$$\bigcup \{ Z_i Z_{16+i} + Z_i Z_{31+i} + Z_{16+i} Z_{31+i} + Z_{32+i} \mid i = 1, \ldots, 15 \} \tag{8}$$

$$\bigcup \{ Z_i + Z_{16+i} + Z_{32+i} + Z_{48+i} \mid i = 1, \ldots, 15 \} \tag{9}$$

$$\bigcup \{ Z_0 \cdot Z_{16} + Z_{32}, Z_0 + Z_{16} + Z_{48}, Z_{63} + Y_2 \} \tag{10}$$

Complexity of other opcodes The opcodes described above, applied to w-bit registers, require $O(w)$ constraints and auxiliary variables (R_{XOR} requires $O(1)$ constraints and auxiliary variables). All other opcodes of the TinyRAM assembly specification [16] can be implemented with $O(w)$ complexity. For most opcodes this can be verified by inspection. For integer multiplication — i.e., to prove that

$$\left(\sum_{i=0}^{w-1} a_i 2^i \right) \cdot \left(\sum_{i=0}^{w-1} b_i 2^i \right) = \sum_{i=0}^{2w-2} c_i 2^i, \quad a_i, b_i, c_i \in \{0, 1\}$$

we fix a generator g for the multiplicative group of $\mathbb{F}$ (the order of g is $2^{63} - 1$ for our choice of field) and then apply repeated squaring to verify that

$$\left(g^{\left(\sum_i a_i 2^i \right)} \right)^{\left(\sum_i b_i 2^i \right)} = g^{\left(\sum_i c_i 2^i \right)}$$

Inspection reveals this solution scales asymptotically like $O(w)$ and for small values, R_{MUL} is twice as costly as R_{ADD} in terms of number of constraints and auxiliary variables.

B.2 Program flow via multi-linear Lagrange polynomials

A program $\mathbb{P}$ of length s is a sequence of instructions $\mathsf{I}_0, \ldots, \mathsf{I}_{s-1}$, each instruction contains an opcode and a list of k inputs and ℓ outputs, where k and ℓ should match the number of inputs and outputs consumed and produced by the opcode, respectively. An input is either a constant (also known as immediate) or a register location and outputs are invariably register locations. (Instructions related to random access memory are dealt with separately, below; until then we assume our programs do not access it and use only the 16 registers.) Each instruction also points to the next instruction in the program; by default I_j points to I_{j+1} but certain instructions (jumps and conditional jumps) may point to a different instruction, and the pointer may further depend on the value of certain registers. The *program counter* (PC) is a special register that contains the number of the current instruction, and thus takes values in $\{0, \ldots, s-1\}$.

A *machine state* is a pair $S = (\mathbf{PC}, \mathbf{R})$ where $\mathbf{PC}$ holds the value of the program counter and $\mathbf{R}$ contains the values of all registers. The program $\mathbb{P}$ induces a natural relation $R_{\mathbb{P}}$ that contains all pairs $(S = (\mathbf{PC}, \mathbf{R}), S' = (\mathbf{PC}', \mathbf{R}'))$ of machine states such that a single cycle of the machine in state S (with program counter being $\mathbf{PC}$ and registers holding values $\mathbf{R}$) results in state S'. As done for opcodes in (1), our purpose in this subsection is to define a system of constraints, denoted $A_{\mathbb{P}}$, that defines $R_{\mathbb{P}}$ as its zero-locus, projected onto its first few variables. Formally, let $\mathbf{PC}, \mathbf{PC}', \mathbf{R}, \mathbf{R}'$ denote variables ranging over $\mathbb{F}$, and recall $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ denote variables for opcode inputs, outputs and auxiliary variables, respectively. Then

$$R_{\mathbb{P}} = \left\{ \left((\mathbf{PC}, \mathbf{R}), (\mathbf{PC}', \mathbf{R}') \right) \mid \exists \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}\, A_{\mathbb{P}}\left(\mathbf{PC}, \mathbf{R}, \mathbf{PC}', \mathbf{R}', \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \right) = 0 \right\} \quad (11)$$

In words, $A_{\mathbb{P}}$ is a set of polynomials whose zero-locus, projected to $\mathbf{PC}, \mathbf{R}, \mathbf{PC}', \mathbf{R}'$, equals the "program evolution" relation $R_{\mathbb{P}}$.

To minimize degree complexity, the program counter value is recorded via $r = \lceil \log s \rceil$ many variables, denoted $\mathsf{PC}_1, \ldots, \mathsf{PC}_r$, each ranging over $\{0, 1\}$. For $\alpha \in \{0, 1\}^r$ let

$$L_\alpha(\mathsf{PC}_1, \ldots, \mathsf{PC}_r) = \prod_{i=1}^{r} (\mathsf{PC}_i + \alpha_i + 1)$$

be the Lagrange multi-linear polynomial that evaluates to 1 on α and evaluates to 0 on $\{0, 1\}^r \setminus \{\alpha\}$. We multiply the polynomials in the algebraic constraint system appearing in the ith instruction by $L_{\bar{i}}(\mathsf{PC}_1, \ldots, \mathsf{PC}_r)$ where $\bar{i} \in \{0, 1\}^r$ is the binary representation of i. Informally, this has the effect of applying the set of constraints A_{op} only when the PC points to an instruction that contains op. Formally, for each opcode op appearing in the program $\mathbb{P}$, let

$I_{\mathsf{op} \in \mathbb{P}} \subseteq \{0, \ldots, s-1\}$ be the set of program instructions in which op is executed. Then define

$$\hat{A}^{\mathsf{op} \in \mathbb{P}} = \left\{ P \cdot \sum_{i \in I_{\mathsf{op}}} L_{\bar{i}}(\mathsf{PC}_1, \ldots, \mathsf{PC}_r) \mid P \in A_{\mathsf{op}} \right\} \tag{12}$$

Inputs and outputs to an opcode are checked in a similar way. In particular, let $\mathsf{i}_{i,1}, \ldots, \mathsf{i}_{i,k_i}$ denote the indices of the registers that are the inputs of the opcode in instruction i and let $\mathsf{o}_{i,1}, \ldots, \mathsf{o}_{i,\ell_i}$ be the indices of output registers of that instruction, then we define

$$\hat{A}_i^{\mathsf{i/o}} = \left\{ (X_j - \mathsf{R}_{\mathsf{i}_{i,j}}) \cdot L_{\bar{i}}(\mathsf{PC}_1, \ldots, \mathsf{PC}_r) \mid j = 1, \ldots, k_i \right\} \tag{13}$$
$$\bigcup \left\{ (Y_j - \mathsf{R}'_{\mathsf{o}_{i,j}}) \cdot L_{\bar{i}}(\mathsf{PC}_1, \ldots, \mathsf{PC}_r) \mid j = 1, \ldots, \ell_i \right\}$$
$$\bigcup \left\{ (\mathsf{R}_j - \mathsf{R}'_j) \cdot L_{\bar{i}}(\mathsf{PC}_1, \ldots, \mathsf{PC}_r) \mid j \text{ is not an output register of instruction } i \right\}$$

In similar fashion, updating the program counter during the ith instruction is defined using a set of polynomials whose zero locus corresponds to the correct update of PC value. Typically, this modification simply increments the value of the PC by 1, and this can be done by multiplying each polynomial in (6)–10) by $L_{\bar{i}}(\mathsf{PC}_1, \ldots, \mathsf{PC}_r)$. Let $\hat{A}_i^{\mathsf{pc}}$ denote the corresponding set of polynomials. The final set $A_{\mathbb{P}}$ that defines the "program evolution" relation $R_{\mathbb{P}}$ is

$$A_{\mathbb{P}} \triangleq \left\{ \hat{A}_{\mathsf{op} \in \mathbb{P}} \mid \mathsf{op} \text{ appears in } \mathbb{P} \right\} \bigcup \left\{ \hat{A}_i^{\mathsf{i/o}} \mid i = 0, \ldots, s-1 \right\} \tag{14}$$
$$\bigcup \left\{ \hat{A}_i^{\mathsf{pc}} \mid i = 0, \ldots, s-1 \right\}$$

and the discussion above shows that its zero locus $A_{\mathbb{P}}$, projected to $\mathbf{PC}, \mathbf{R}, \mathbf{PC'}, \mathbf{R'}$, indeed equals $R_{\mathbb{P}}$.

C Two programs computing subset-sum

Code 1 shows a high-level description of the exhaustive subset-sum program, and Code 2 gives an equivalent TinyRAM hand-optimized implementation (cf. Appendix D for discussion of machine compiled assembly). In Code 1, the variable k is treated as a binary vector that iterates over all the possible combinations of the inputs. The inputs that correspond to each combination are summed up by inspecting whether the least significant bit (LSB) of k is 1, and then shifting k rightward. Code 2 uses the AND,CMPE,SHR TinyRAM instructions for these inspections and shifts. It should be noted that the instruction set that is needed for Code 2 is uncostly, in particular the cost of the DIV instruction would have been about twice higher than SHR in terms of the number of field elements that the prover commits to in a time step.

The total number of time steps T of the ACSP for Code 2 is sufficiently large if the inequality $2^n \cdot (9n + 7) < T$ holds, where n is the size of the input array. With 16-bit TinyRAM architecture, $n \leq 16$ is also required, unless extra logic is added to Code 2. In this inequality, the term $9n$ can be inferred by amortizing the number of TinyRAM instructions that are executed when the LSB of k is either 0 or 1. For example, $T = 2^{20}$ is sufficient for $n = 13$ inputs. For a further demonstration of the dependency between T and n, see Figure 2.

The TinyRAM architecture relies on 16 or less registers, in particular Code 2 needs 5 registers in total. This helps with keeping the complexity low, as it implies that a relatively small number of field elements are required per time step. However, this also means that we do not have enough registers to store the entire input array. Since it is preferable to avoid the poly-logarithmic blowup of programs with memory, Code 2 employs a special "read-only memory" (ROM) instruction. The ROM instruction takes a single operand, treats it as an index $J \leq n$, and returns the corredponding array$[J]$ input value. The algebraic constraints of the ROM instruction consist of unpacking the bits of J and using a selector polynomial to force the prover to use the predefined array$[J]$ field element. For example, with $n = 8$, the ROM instruction can be implemented as

$$\bigcup_{k=0}^{2} \{b_k(b_k+1)\} \bigcup \{J + \sum_{k=0}^{2} b_k x^k, \sum_{\alpha,\beta,\gamma \in \{0,1\}} (b_0+\alpha)(b_1+\beta)(b_2+\gamma)(R+C_{\alpha,\beta,\gamma})\},$$

where R is the returned operand and $C_{\alpha,\beta,\gamma}$ are the array input values that the ACSP instance specifies. Thus, the degree of the ROM constraints is bounded by $\lceil \log n \rceil + 1$, and overall the ROM instruction is far less complex than deploying the full read/write memory construction.

Code 3 is a subset-sum program that computes all the partial sums of half of the input numbers, as well as the other half, and then does a linear scan to look for two partial sums that add up to the target value [51]. The partial sums are first stored in memory in a sorted order, which can be done in $O(n)$ time due to the following observation: given a sorted list $S_1, S_2, \ldots, S_{2^k}$ of all the possible sums that can be produced from combinations of certain k numbers, and another number m, the sorted list $S_1 + m, S_2 + m, \ldots, S_{2^k} + m$ can be merged into $S_1, S_2, \ldots, S_{2^k}$ to obtain one sorted list of size 2^{k+1}, in linear time. Hence, Code 3 needs to store $O(\sqrt{2^n})$ elements in memory, where n is the size of the input array.

Code 4 gives a hand-optimized TinyRAM implementation of this high-level pseudocode, in which the dependecy between n and the total number of time steps T is $n \approx 2(T - 7)$. Section D discusses the machine compiled code for the same program. As can be seen in Figure 2, Code 4 can thus cope with greater values of n than Code 2, even after the poly-logarithmic blowup in complexity that is due to memory handling is taken into account.

Notice that unlike the high-level description in Code 3, the Code 4 implementation that we benchmark actually outputs a bit-string of the correct combination, if one exists (Code 5 and Code 6 do this as well). This extra work is done for a

fair comparison with Code 2, that does this "for free". However, since subset-sum is an NP-complete problem, it makes sense to generate the PCP on unsatisfiable instances. Thus, this extra work can be regarded as unnecessary in this context.

Code 1   Pseudocode of the exhaustive search subset-sum program

---

input: $n, \texttt{array}[n], \texttt{target}$
1: **for** $k = 1$ **to** $2^n - 1$ **do** $\triangleright$ k loops over all $\{0,1\}^n \setminus \{0^n\}$ combinations
2:    $curr \leftarrow k,\ idx \leftarrow 0,\ sum \leftarrow 0$
3:    **while** $curr \neq 0$ **do**
4:       **if** $1 = (curr\ \textbf{bitwise-and}\ 1)$ **then** $\triangleright$ LSB of $curr$ is 1?
5:          $sum \leftarrow sum + \texttt{array}[idx]$
6:       **end if**
7:       $curr \leftarrow curr/2,\ idx \leftarrow idx + 1$
8:    **end while**
9:    **if** $sum = \texttt{target}$ **then**
10:       **return** k
11:    **end if**
12: **end for**
13: **return** 0

Code 2   TinyRAM assembly code of the exhaustive search subset-sum program

---

```
 1: MOV    r0, 1          9: CJMP   Line#12       17: CMPE   r1, target
 2: CMPE   r0, 2^n       10: ROM    r4, r3        18: CJMP   Line#22
 3: CJMP   Line#21       11: ADD    r1, r1, r4    19: ADD    r0, r0, 1
 4: MOV    r1, 0         12: SHR    r2, r2, 1     20: JMP    Line#2
 5: MOV    r2, r0        13: CMPE   r2, 0         21: MOV    r0, 0
 6: MOV    r3, 0         14: CJMP   Line#17       22: ANSW   r0
 7: AND    r4, r2, 1     15: ADD    r3, r3, 1
 8: CMPE   r4, 0         16: JMP    Line#7
```

D   Compiling C code to TinyRAM

Our TinyRAM compiler is implemented as a `GCC` back end, with support for some optimization techniques. Code 5 shows C source for the memory-based subset-sum program, and the corresponding compiled code is given as Code 6. As shown, Code 6 has 21 more instruction than the hand-written assembly of Code 4. Likewise, the running time of Code 6 is somewhat greater than that of Code 4, for example with $n = 14$ it takes 13582 time steps until Code 6 terminates, while Code 4 terminates in 11231 time steps.

Code 3   Pseudocode of the memory-based subset-sum program

---

input: $n = 2h, \mathtt{array}[n], \mathtt{target}$

```
 1: H_1 ← {array[0], array[1], ..., array[h − 1]}
 2: H_2 ← {array[h], array[1], ..., array[n − 1]}
 3: for m ∈ {1, 2}  do                              ▷ sort each half
 4:     let A_{m,0} be an array of size 1 with A_{m,0}[0] = 0
 5:     i ← 0
 6:     for x ∈ H_m  do
 7:         let B_{m,i} be an array of size i and C_{m,i} be an array of size 2i
 8:         for k ∈ {0, 1, 2, ..., 2^i − 1}  do
 9:             B_{m,i}[k] ← A_{m,i}[k] + x
10:         end for
11:         C_{m,i} ← merge(A_{m,i}, B_{m,i})      ▷ note: A_{m,i} and B_{m,i} are already sorted
12:         A_{m,i+1} ← C_{m,i}
13:         i ← i + 1
14:     end for
15: end for
16: i ← 0,  k ← 2^h − 1
17: while True  do                                  ▷ search for the target
18:     if target = A_{1,h}[i] + A_{2,h}[k] then return 1 end if
19:     if target > A_{1,h}[i] + A_{2,h}[k] then
20:         if i = 2^h − 1 then return 0 end if
21:         i ← i + 1
22:     else
23:         if k = 0 then return 0 end if
24:         k ← k − 1
25:     end if
26: end while
```

Code 4   TinyRAM assembly code of the memory-based subset-sum program

input: $n = 2h$, `array[n]`, `target`, $\ell = 2^{h+1} - 2$
constants: $\texttt{INPADDR} = 2^{16} - 2^6, \texttt{ADDR1} = 0, \texttt{ADDR2} = 2^{14}, \texttt{OFFSET} = 2^{15}$
preprocess: store `array[n]` at `INPADDR`

```
 1: MOV    r0, INPADDR        31: ADD    r6, r2, OFFSET     61: CJMP   Line#64
 2: MOV    r1, ADDR1          32: LOAD   r6, r6             62: MOV    r1, ADDR2
 3: MOV    r9, 0              33: XOR    r6, r6, r8         63: JMP    Line#3
 4: STOR   r9, r1             34: ADD    r2, r2, 1          64: MOV    r0, ADDR1 + ℓ
 5: ADD    r2, r1, OFFSET     35: JMP    Line#21            65: LOAD   r2, r0
 6: STOR   r9, r2             36: CMPE   r5, r2             66: LOAD   r3, r1
 7: MOV    r2, r1             37: CNJMP  Line#44            67: ADD    r4, r2, r3
 8: ADD    r4, r1, 1          38: LOAD   r6, r1             68: CMPE   r4, target
 9: MOV    r5, r4             39: STOR   r6, r4             69: CJMP   Line#L83
10: MOV    r8, 1              40: ADD    r6, r1, OFFSET     70: CMPG   r4, target
11: ADD    r9, h              41: LOAD   r6, r6             71: CJMP   Line#77
12: LOAD   r3, r0             42: ADD    r1, r1, 1          72: CMPE   r1, ADDR2 + ℓ
13: JMP    Line#44            43: JMP    Line#21            73: CJMP   Line#82
14: ADD    r0, r0, 1          44: LOAD   r6, r1             74: ADD    r1, r1, 1
15: CMPE   r9, r0             45: LOAD   r7, r2             75: LOAD   r3, r1
16: CJMP   Line#60            46: ADD    r7, r7, r3         76: JMP    Line#67
17: LOAD   r3, r0             47: CMPG   r6, r7             77: CMPE   r0, ADDR1
18: SHL    r8, r8, 1          48: CJMP   Line#54            78: CJMP   Line#82
19: MOV    r5, r4             49: STOR   r6, r4             79: SUB    r0, r0, 1
20: JMP    Line#44            50: ADD    r6, r1, OFFSET     80: LOAD   r2, r0
21: ADD    r7, r4, OFFSET     51: LOAD   r6, r6             81: JMP    Line#67
22: STOR   r6, r7             52: ADD    r1, r1, 1          82: ANSW   0
23: ADD    r4, r4, 1          53: JMP    Line#21            83: ADD    r2, r0, OFFSET
24: CMPE   r5, r1             54: STOR   r7, r4             84: LOAD   r2, r2
25: CNJMP  Line#36            55: ADD    r6, r2, OFFSET     85: ADD    r3, r1, OFFSET
26: CMPE   r5, r2             56: LOAD   r6, r6             86: LOAD   r3, r3
27: CJMP   Line#14            57: XOR    r6, r6, r8         87: SHL    r3, r3, H
28: LOAD   r6, r2             58: ADD    r2, r2, 1          88: XOR    r2, r2, r3
29: ADD    r6, r6, r3         59: JMP    Line#21            89: ANSW   r2
30: STOR   r6, r4             60: CMPA   r1, ADDR2
```

Code 5   C source of the memory-based subset-sum program

```c
#define N 7
#define TARGET 123

int input[2*N] = {10,20,30,40,50,60,70,-10,-20,-30,-40,-50,-60,70};
int arr[ 4 * ( (1 << (N+1)) - 1 ) ];

int main(void) {
    register int *inp = &input[0], *last_inp, *p1, *p2, *next, *next_backup, b;
    p1 = p2 = &arr[0]; //phase1: prepare arrays
    for(;;) { //prepare each half array
        next = next_backup = (p1+2);
        *p1 = *(p1+1) = 0; b = 1; last_inp = inp + N;
        for(;;) { //iterate over each input
            for(;;) { //merge
                if(p1 == next_backup) {
                    while(p2 < next_backup) {
                        *(next++) = *(p2++) + *inp;
                        *(next++) = *(p2++) ^ b;
                    }
                    break;
                }
                if(p2 == next_backup) {
                    while(p1 < next_backup) {
                        *(next++) = *(p1++);
                        *(next++) = *(p1++);
                    }
                    break;
                }
                if(*p1 > *p2 + *inp) {
                    *(next++) = *(p2++) + *inp;
                    *(next++) = *(p2++) ^ b;
                }
                else {
                    *(next++) = *(p1++);
                    *(next++) = *(p1++);
                }
            }
            if(++inp == last_inp) break;
            b = b << 1;
            next_backup = next;
        }
        if( p1 > &arr[0] + (1 << (N+2)) ) break;
        p1 = p2 = next;
    }
    p1 = &arr[ 2*((1 << (N+1)) - 1) - 2 ]; //phase2: search
    for(;;) {
        if(TARGET == *p1 + *p2)
            return *(p1+1) ^ (*(p2+1) << N);
        if(TARGET > *p1 + *p2) {
            if(p2 == &arr[0] + 4*((1 << (N+1))-1) - 2) break;
            p2 = p2 + 2;
        }
        else {
            if(p1 == &arr[0])    break;
            p1 = p1 - 2;
        }
    }
    return 0;
}
```

Code 6   TinyRAM assembly code of the compiled subset-sum program

input: $n = 2h$,   `array[n]`,   `target`
preprocess: store `array[n]` at address 0

```
 1: MOV    r9, 0           38: LOAD   r2, r8          75: SHL    r14, r14, 1
 2: MOV    r12, 28         39: ADD    r8, r8, 2       76: MOV    r13, r4
 3: MOV    r8, r12         40: STOR   r2, r4          77: JMP    Line#12
 4: ADD    r13, r8, r4     41: ADD    r4, r4, 2       78: CMPA   r8, 1052
 5: MOV    r4, r13         42: CMPAE  r8, r13         79: CJMP   Line#83
 6: MOV    r2, 0           43: CNJMP  Line#34         80: MOV    r12, r4
 7: ADD    r0, r8, 2       44: JMP    Line#72         81: MOV    r8, r4
 8: STOR   r2, r0          45: LOAD   r2, r12         82: JMP    Line#4
 9: STOR   r2, r8          46: LOAD   r3, r9          83: MOV    r4, 1044
10: MOV    r14, 1          47: ADD    r3, r2, r3      84: LOAD   r3, r4
11: ADD    r5, r9, 14      48: LOAD   r2, r8          85: LOAD   r2, r12
12: CMPE   r8, r13         49: CMPG   r2, r3          86: ADD    r2, r3, r2
13: CNJMP  Line#30         50: CNJMP  Line#63         87: CMPE   r2, target
14: CMPAE  r12, r13        51: LOAD   r3, r12         88: CNJMP  Line#96
15: CJMP   Line#72         52: LOAD   r2, r9          89: ADD    r0, r12, 2
16: LOAD   r3 r12          53: ADD    r2, r3, r2      90: LOAD   r12, r0
17: LOAD   r2, r9          54: ADD    r12, r12, 2     91: SHL    r12, r12, h
18: ADD    r2, r3, r2      55: STOR   r2, r4          92: ADD    r0, r4, 2
19: ADD    r12, r12, 2     56: ADD    r4, r4, 2       93: LOAD   r4, r0
20: STOR   r2, r4          57: LOAD   r2, r12         94: XOR    r2, r12, r4
21: ADD    r4, r4, 2       58: XOR    r2, r14, r2     95: JMP    Line#110
22: LOAD   r2, r12         59: ADD    r12, r12, 2     96: LOAD   r3, r4
23: XOR    r2, r14, r2     60: STOR   r2, r4          97: LOAD   r2, r12
24: ADD    r12, r12, 2     61: ADD    r4, r4, 2       98: ADD    r2, r3, r2
25: STOR   r2, r4          62: JMP    Line#12         99: CMPG   r2, target-1
26: ADD    r4, r4, 2       63: LOAD   r2, r8         100: CJMP   Line#106
27: CMPAE  r12, r13        64: ADD    r8, r8, 2      101: MOV    r2, 2064
28: CNJMP  Line#16         65: STOR   r2, r4         102: CMPE   r12, r2
29: JMP    Line#72         66: ADD    r4, r4, 2      103: CJMP   Line#109
30: CMPE   r12, r13        67: LOAD   r2, r8         104: ADD    r12, r12, 4
31: CNJMP  Line#45         68: ADD    r8, r8, 2      105: JMP    Line#84
32: CMPAE  r8, r13         69: STOR   r2, r4         106: CMPE   r4, 28
33: CJMP   Line#72         70: ADD    r4, r4, 2      107: CJMP   Line#109
34: LOAD   r2, r8          71: JMP    Line#12        108: JMP    Line#84
35: ADD    r8, r8, 2       72: ADD    r9, r9, 2      109: MOV    r2, 0
36: STOR   r2, r4          73: CMPE   r9, r5         110: ANSW   r2
37: ADD    r4, r4, 2       74: CJMP   Line#78
```