Arya: Nearly Linear-Time Zero-Knowledge Proofs for Correct Program Execution

Jonathan Bootle(✉), Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller

University College London, London, UK
{jonathan.bootle.14,andrea.cerulli.13,j.groth,s.jakobsen, mary.maller.15}@ucl.ac.uk

Abstract. There have been tremendous advances in reducing interaction, communication and verification time in zero-knowledge proofs but it remains an important challenge to make the prover efficient. We construct the first zero-knowledge proof of knowledge for the correct execution of a program on public and private inputs where the prover computation is nearly linear time. This saves a polylogarithmic factor in asymptotic performance compared to current state of the art proof systems.

We use the TinyRAM model to capture general purpose processor computation. An instance consists of a TinyRAM program and public inputs. The witness consists of additional private inputs to the program. The prover can use our proof system to convince the verifier that the program terminates with the intended answer within given time and memory bounds. Our proof system has perfect completeness, statistical special honest verifier zero-knowledge, and computational knowledge soundness assuming linear-time computable collision-resistant hash functions exist. The main advantage of our new proof system is asymptotically efficient prover computation. The prover's running time is only a superconstant factor larger than the program's running time in an apples-to-apples comparison where the prover uses the same TinyRAM model. Our proof system is also efficient on the other performance parameters; the verifier's running time and the communication are sublinear in the execution time of the program and we only use a log-logarithmic number of rounds.

Keywords: Zero-knowledge proofs
Succinct arguments of knowledge · TinyRAM
Ideal linear commitments · Post-quantum security

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013)/ERC Grant Agreement n. 307937.

M. Maller—Supported by a scholarship from Microsoft Research.

T. Peyrin and S. Galbraith (Eds.): ASIACRYPT 2018, LNCS 11272, pp. 595–626, 2018.
https://doi.org/10.1007/978-3-030-03326-2_20

1 Introduction

A zero-knowledge proof system [GMR85] enables a prover to convince a verifier that a statement is true without revealing anything else. We are interested in proving statements of the form $u \in \mathcal{L}$, where $\mathcal{L}$ is a language in NP. A zero-knowledge proof is an interactive protocol between a prover and a verifier, where both hold the same instance u, and the prover also holds a witness w to $u \in \mathcal{L}$. The protocol should satisfy three properties:

Completeness: A prover holding a witness to $u \in \mathcal{L}$ can convince the verifier.
Soundness: A cheating prover cannot convince the verifier when $u \notin \mathcal{L}$.
Zero-knowledge: The interaction only shows the statement $u \in \mathcal{L}$ is true. It reveals nothing else, in particular it does not disclose anything of the witness.

Zero-knowledge proofs have numerous applications and are for instance used in constructions of public-key encryption schemes secure against chosen ciphertext attack, digital signatures, voting systems, auction systems, e-cash, secure multi-party computation, and verifiable outsourced computation. The zero-knowledge proofs impact the performance of all these applications, and it is therefore important for them to be as efficient as possible.

There are many zero-knowledge proofs for dealing with arithmetic or boolean circuit satisfiability. However, in applications usually the type of statements we want to prove is that a protocol participant is following the protocol honestly; whatever that protocol may be. This means we want to express statements relating to program execution such as "running program P specified by the protocol on public input x and private input y returns the output z." In principle such a statement can be reduced to circuit satisfiability but the cost of the NP-reduction incurs a prohibitive cost. In this paper, we therefore focus on the important question of getting zero-knowledge proofs for statements relating directly to program execution.

Performance can be measured on a number of parameters including the prover's running time, the verifier's running time, the number of transmitted bits and the number of rounds the prover and verifier interact. Current state of the art zero-knowledge proofs get very good performance on verification time, communication and round complexity, which makes the prover's running time the crucial bottleneck. Indeed, since the other costs are so low, we would happily increase them for even modest savings on the proving time since this is the barrier that make some applications such as verifiable outsourced computation currently unviable. The research challenge we focus on is therefore to get *prover-efficient* zero-knowledge proofs for correct program execution.

1.1 Our Contribution

We use the TinyRAM model [BCG+13, BSCG+13] for computation. TinyRAM specifies a random access machine with a small instruction set working on W-bit words and addresses. The specification of TinyRAM considers a Harvard-architecture processor, which means that the program being executed is stored

separately from the data being processed and does not change during execution.[1] Experimental results [BCG+13] show that programs written in C can be compiled efficiently into TinyRAM programs and only have a modest constant overhead compared to optimized compilation to machine code on a modern processor.

In our proof system, an instance consists of a TinyRAM program and public data given to the program, and a witness is private data given as input to the program. The statement is the claim that the TinyRAM program P running on given public and private data will terminate with answer 0 within specific time and memory bounds. When measuring performance we think of the prover and verifier as being TinyRAM programs with the same word size[2].

Our main contribution is an interactive proof system for correct TinyRAM computation, which has perfect completeness, statistical zero-knowledge, and computational knowledge soundness based on collision-resistant hash functions. Knowledge soundness means that not only do we have soundness and it is infeasible to prove a false statement, but it is also a proof of knowledge such that given access to a successful prover it is possible to extract a witness. For maximal asymptotic efficiency we may use linear-time computable hash functions, which yields the performance given in Fig. 1.

Our proof system is highly efficient for computationally intensive programs where the execution time dominates other parameters (see Sect. 6 for a detailed discussion of parameter choices). For a statement about the execution of a TinyRAM program of length L, running with time bound T and memory bound M, the prover runs in $\mathcal{O}(\alpha T)$ steps[3] for an arbitrarily small superconstant function $\alpha(\lambda) = \omega(1)$. The proof system is also efficient on other performance parameters: the verifier running time and the communication grows roughly with the square-root of the execution time[4] and we have log-logarithmic round complexity. Figure 1 gives an efficiency comparison with a state of the art zk-SNARK [BCTV14b] for verifying correct program execution on TinyRAM.

[1] TinyRAM can with minor changes also be adapted to a von Neumann architecture where program instructions are fetched from memory [BCTV14b]. The performance of our proof systems adapted to a von Neumann architecture would remain the same up to a constant factor.

[2] We stress the choice of comparing the prover and verifier to program execution on the same platform. We do this to get an apples-to-apples comparison; there are many zero-knowledge proofs that are "linear time" because they use different metrics for statement evaluation and the prover time, for instance that the cost of validating the statement given the witness is measured in field multiplications and the prover computation is measured in exponentiations.

[3] The big-O notation hides big constants and we do not recommend implementing the proof system as it is; our contribution is to make significant *asymptotic* gains compared to state-of-the-art zero-knowledge proofs by demonstrating that the prover's computation can be nearly linear.

[4] Disregarding the SHVZK property for a moment, this is also the first proof system for general purpose computation that has both nearly linear computation for the prover and sublinear communication.

Further discussion of other proof systems that can verify correct TinyRAM or other types of program execution can be found in Sect. 1.3. The best of these achieve similar asymptotic prover efficiency as [BCTV14b].

Work	Prover	Verifier	Communication	Rounds	Assumption
[BCTV14b]	$\Omega(T \log^2 T)$	$\omega(L + \|v\|)$	$\omega(1)$	1	KoE
This work	$\mathcal{O}(\alpha T)$	$\text{poly}(\lambda)(\sqrt{T} + L + \|v\|)$	$\text{poly}(\lambda)(\sqrt{T} + L)$	$\mathcal{O}(\log \log T)$	LT-CRHF

Fig. 1. Efficiency comparisons between our arguments and the most efficient zero-knowledge argument for the correct execution of TinyRAM programs, both at security level $2^{-\omega(\log \lambda)}$. Computation is measured in TinyRAM steps and communication in words of length $W = \Theta(\log \lambda)$ with λ the security parameter. KoE stands for knowledge of exponent type assumption in pairing-based groups and LT-CRHF stands for linear-time collision resistant hash function. It is worth noting KoE assumptions do not resist quantum computers while a LT-CRHF may be quantum resistant.

Remarks. Our proof system assumes some public parameters to be set up that include a description of a finite field, an error-correcting code, and a collision-resistant hash function. The size of the public parameters is just $\text{poly}(\lambda)(L + M + \sqrt{T})$ bits which can be computed from a small uniformly random string in $\text{poly}(\lambda)(L + M + \sqrt{T})$ TinyRAM steps. This means the public parameters have little effect on the overall efficiency of the proof system. Moreover, there are variants of the parameters where it is efficiently verifiable the public parameters have the correct structure. This means the prover does not need to trust the parameters to get special honest verifier zero-knowledge, so they can be chosen by the verifier making our proof systems work in the plain model without setup. We let the public parameter be generated by a separate setup though because they are independent of the instance and can be used over many separate proofs.

We did not optimize communication and verification time to go below $\sqrt{T}$ but if needed it is possible to compose our proof system with a verifier-efficient proof system and get verification time that grows logarithmically in T. This is done by letting the prover send linear-time computable hashes of her messages to the verifier instead of the full messages. Since our proof system is public coin the prover knows after this interaction exactly how the verifier in our proof system ought to run if given the messages in our proof system. She can therefore give a verifier-efficient proof of knowledge that she knows pre-images to the hashes that would make the verifier in our proof system accept. We outline this procedure in the full paper [BCG+18].

1.2 New Techniques

Ben-Sasson et al. [BCG+13, BCTV14b] offer proof systems for correct TinyRAM program execution where the prover commits to a time-sorted execution trace as well as an address-sorted memory trace. They embed words, addresses and

flags that describe the TinyRAM state at a given time into field elements. The correct transition in the execution trace between the state at time t and the state at time $t + 1$ can then be checked by an arithmetic circuit, the correct writing and reading of memory at a particular address in the memory trace can be checked by another arithmetic circuit, and finally the consistency of memory values in the two traces can be checked by a third arithmetic circuit that embeds a permutation network. Importantly, in these proofs the state transitions can be proved with the same arithmetic circuits in each step so many of the proofs can be batched together at low average cost.

Combining their approach with the recent linear-time proofs for arithmetic circuit satisfiability by Bootle et al. [BCG+17] it would be possible to get a zero-knowledge proof system with sublinear communication and efficient verification. The prover time, however, would incur at least a logarithmic overhead compared to the time to execute the TinyRAM program. First, the use of an arithmetic circuit that embeds a permutation network to check consistency between execution and memory traces requires a logarithmic number of linear-size layers to describe an arbitrary permutation which translates into a logarithmic overhead when generating the proof. Second, TinyRAM allows both arithmetic operations such as addition and multiplication of words, and logical operations such as bitwise XOR, AND and OR. To verify logical operations they decompose words into single bits that are handled individually. Bit-decomposition makes it easy to implement the logical operations, but causes an overhead when embedding bits into full size field elements. From a technical perspective our main contribution is to overcome these two obstacles.

To reduce the time required to prove the execution trace is consistent with the memory usage we do not embed a permutation network into an arithmetic circuit. Instead we relate memory consistency to the existence of a permutation that maps one memory access in the execution trace to the next access of the same memory address in the execution trace. Neff [Nef01] proposed permutation proofs in the context of shuffle proofs used in mix-nets. Follow-up works [Gro10b, GI08] have improved efficiency of such proofs with Bayer and Groth [BG12] giving a shuffle argument in the discrete logarithm setting where the prover uses a linear number of exponentiations and communication is sublinear. These shuffle proofs are proposed for the discrete logarithm setting and we do not want to pay the cost of computing exponentiations. The core of the shuffle proofs can be formulated abstractly using homomorphic commitments to vectors though. Since the proofs by Bootle et al. [BCG+17] also rely on an idealization of homomorphic commitments to vectors the ideas are compatible and we get permutation proofs that cost a linear number of field operations.

To remove the overhead of bit-decomposition we invent a less costly decomposition. While additions and multiplications are manageable using a natural embedding of words into field elements, such a representation is not well suited to logical operations though. However, instead of decomposing words into individual bits, we decompose them into interleaved odd-position bits and even-position bits. A nibble (a_3, a_2, a_1, a_0) can for instance be decomposed into $(a_3, 0, a_1, 0) + (0, a_2, 0, a_0)$. The key point of this idea is that adding two interleaved even bit

nibbles yields $(0, a_2, 0, a_0) + (0, b_2, 0, b_0) = (a_2 \wedge b_2, a_2 \oplus b_2, a_0 \wedge b_0, a_0 \oplus b_0)$. So using another decomposition into odd-position and even-position bits we can now extract the XORs and the ANDs. Using this core idea, it is possible to represent all logical operations using field additions together with decomposition into odd and even-position bits. This reduces the verification of logical operations to verifying correct decomposition into odd and even bits. This decomposition and its use are described in the full paper [BCG+18].

To enable decomposition proofs into odd and even-position bits, we develop a new lookup proof that makes it possible to check that a field element belongs to a table of permitted values. By creating a lookup table of all words with even-position bits, we make it possible to verify such decompositions. Lookup proofs not only enable decomposition into odd and even-position bits but also turn out to have many other uses such as demonstrating that a field element represents a correct program instruction, or that a field element represents a valid word within the range $\{0, \ldots, 2^W - 1\}$.

Combining arithmetic circuits, permutations and table lookups we get a set of conditions for a TinyRAM execution being correct. The program execution of T steps on the TinyRAM machine can in our system be encoded as $\mathcal{O}(T)$ field elements that satisfy the conditions. Using prime order fields of size $2^{\mathcal{O}(W)}$ would make it possible to represent these field elements as $\mathcal{O}(1)$ words each. However, the soundness of our proof systems depends on the field size and to get negligible soundness error we choose a larger field to get a superconstant ratio $e = \frac{\log |\mathbb{F}|}{W}$. This factors into the efficiency of our proof system giving a prover runtime of $\mathcal{O}(\alpha T)$ TinyRAM steps for an instance requiring time T, where α is a superconstant function which specifies how many steps it takes to compute a field operation, i.e., $\alpha = \mathcal{O}(e^2)$.

Having the inner core of conditions in place: arithmetic circuits for instruction executions, permutations for memory consistency, and look-ups for word decompositions we now deploy the framework of Bootle et al. [BCG+17] to get a zero-knowledge proof system. They use error-correcting codes and linear-time collision-resistant hash functions to give proof systems for arithmetic circuit satisfiability, while we will use their techniques to prove our conditions on the execution trace are satisfied. Their proof system for arithmetic circuit satisfiability requires the prover to use a linear number of field multiplications and the verifier to use a linear number of field additions. However, we can actually get sublinear verification when the program and the input is smaller than the execution time. Technically, the performance difference stems from the type of permutation proof that they use for verifying the correct wiring of the circuit and that we use for memory consistency in the execution trace. In their use, the permutation needs to be linked to the publicly known wiring of the arithmetic circuit and in order for the verifier to check the wiring is correct he must read the entire circuit. We on the other hand do not disclose the memory accesses in the execution trace to the verifier, indeed to get zero-knowledge it is essential the memory accesses remain secret. We therefore need a hidden permutation proof and such proofs can have sublinear verification time.

1.3 Related Work

Interaction. Interaction is measured by the number of rounds the prover and verifier exchange messages. Feige and Shamir [FS90] showed that constant round argument systems exist, and Blum, Feldman and Micali [BFM88] showed that if the prover and verifier have access to an honestly generated common reference string it is possible to have non-interactive zero-knowledge proofs where the prover sends a single message to the verifier.

Communication. A series of works [KR08, IKOS09, Gen09, GGI+15] have constructed proof systems where the number of transmitted bits is proportional to the witness size. It is unlikely that sublinear communication is possible in proof systems with statistical soundness but Kilian [Kil92] constructed an argument system, a computationally sound proof system, with polylogarithmic communication complexity. Kilian's zero-knowledge argument relies on probabilistically checkable proofs [AS98], which are still complex for practical use, but the invention of interactive oracle proofs [BCS16] have made this type of proof system a realistic option. Recent work by Ben-Sasson et al. [BSBTHR18] presents a new PCP-based argument system, known as STARKs, which also has polylogarithmic communication costs, and is optimized for better practicality. Ishai et al. [IKO07] give laconic arguments where the prover's communication is minimal. Groth [Gro10a], working in the common reference string model and using strong assumptions, gave a pairing-based non-interactive zero-knowledge argument consisting of a constant number of group elements. Follow-up works on succinct non-interactive arguments of knowledge (SNARKs) have shown that it is possible to have both a modest size common reference string and proofs as small as 3 group elements [BCCT12, GGPR13, PHGR16, BCCT13, Gro16].

Verifier Computation. In general the verifier has to read the entire instance since even a single deviating bit may render the statement $u \in \mathcal{L}$ false. However, in many cases an instance can be represented more compactly than the witness and the instance may be small compared to the computational effort it takes to verify a witness for the instance. In these cases it is possible to get sublinear verification time compared to the time it takes to check the relation defining the language $\mathcal{L}$. This is for instance the case for the SNARKs mentioned above, where the verification time only depends on the size of the instance but not the complexity of the relation.

Prover Computation. Given the success in reducing interaction, communication and verification time, the important remaining challenge is to get good efficiency for the prover.

Boolean and Arithmetic Circuits. Many classic zero-knowledge proofs rely on cyclic groups and have applications in digital signatures, encryption schemes, etc. The techniques first suggested by Schnorr [Sch91] can be generalized

to NP-completel languages such as boolean and arithmetic circuit satisfiability [CD, Gro09, BCC+16]. In these proofs and arguments the prover uses $\mathcal{O}(N)$ group exponentiations, where N is the number of gates in the circuit. For the discrete logarithm assumption to hold, the groups must have superpolynomial size in the security parameter though, so exponentiations incur a significant overhead compared to direct evaluation of the witness in the circuit. The SNARKs mentioned earlier also rely on cyclic groups and likewise require the prover to do $\mathcal{O}(N)$ exponentiations. Recently, Bootle et al. [BCG+17] used the structure of [Gro09] to give constant overhead zero-knowledge proofs for arithmetic circuit satisfiability, where the prover uses $\mathcal{O}(N)$ field multiplications, relying on error-correcting codes and efficient collision-resistant hash functions instead of cyclic groups. STARKs [BSBTHR18] achieve slightly worse, quasilinear prover computation but have lower asymptotic verification costs.

An alternative to these techniques is to use the "MPC in the head" paradigm by Ishai et al. [IKOS09]. Relying on efficient MPC techniques, Damgård, Ishai and Krøigaard gave zero-knowledge arguments with little communication and a prover complexity of polylog$(\lambda)N$. Instead of focusing on theoretical performance, ZKBoo [GMO16] and its subsequent optimisation ZKB++ [CDG+17] are practical implementations of a "3PC in the head" style zero-knowledge proof for boolean circuit satisfiability. Communication grows linearly in the circuit size in both proofs, and a superlogarithmic number of repetitions is required to make the soundness error negligible, but the speed of the symmetric key primitives makes practical performance good. Ligero [AHIV17] provides another implementation using techniques related to [BCG+17]. It has excellent practical performance but asymptotically it is not as efficient as [BCG+17] due to the use of more expensive error-correcting codes. Another alternative also inspired by the MPC world is to use garbled circuits to construct zero-knowledge arguments for boolean circuits [BP12, JKO13, FNO15]. The proofs grow linearly in the size of the circuit and there is a polylogarithmic overhead for the prover and verifier due to the cryptographic operations but implementations are practical [JKO13].

There are several proof systems for efficient verification of outsourced computation [GKR08, CMT12, Tha13, WHG+16]. While this line of works mostly focus on verifying deterministic computation and does not require zero-knowledge, recent works add in cryptographic techniques to obtain zero-knowledge [ZGK+17, WJB+17, WTas+17]. Hyrax [WTas+17] offers an implementation with good concrete performance. It has sublinear communication and verification, while the prover computation is dominated by $\mathcal{O}(dN + S \log S)$ field operations for a depth d and width S circuit when the witness is small compared to the circuit size. If in addition the circuit can be parallelized into many identical sub-computations the prover cost can be further reduced to $O(dN)$ field operations. The system vSQL [ZGK+17] is tailored towards verifying database queries and as in this work it avoids the use of permutation networks using permutation proofs based on invariance of roots in polynomials as first suggested by Neff [Nef01].

Correct Program Execution. In practice, most computation does not resemble circuit evaluation but is instead done by computer programs processing one instruction at a time. There has been a sustained effort to construct efficient zero-knowledge proofs that support real-life computation, i.e., proving statements of the form "when executing program P on public input x and private input y we get the output z." In the context of SNARKs there are already several systems for proving correct execution of programs written in C [PHGR16,BFR+13,BCG+13,WSR+15]. These system generally involve a *front-end* which compiles the program into an arithmetic circuit which is then fed into a cryptographic *back-end*. Much work has been dedicated to improving both sides and achieving different trade-offs between efficiency and expressiveness of the computation.

When we want to reason theoretically about zero-knowledge proofs for correct program execution, it is useful to abstract program execution as a random-access machine that in each instruction can address an arbitrary location in memory and do integer operations on it. For closer resemblance to real-life computation, we can bound the integers to a specific word size and specify a more general set of operations the random-access machine can execute. TinyRAM [BSCG+13, BCG+13] is a prominent example of a computational model bridging the gap between theory and real-word computation. It comes with a compiler from C to TinyRAM code and underpins several implementations of zero-knowledge proofs for correct program execution [BCG+13,BCTV14b,BCTV14a,CTV15, BBC+17] where the prover time is $\Omega(T \log^2 \lambda)$ for a program execution that takes time T. Similar efficiency is also achieved, asymptotically, by other proof systems that can compile (restricted) C programs and prove correct execution such as Pinocchio [PHGR16], Pantry [BFR+13] and Buffet [WSR+15]. Our work reduces the prover's overhead from $\Omega(\log^2 \lambda)$ to an arbitrary superconstant $\alpha = \omega(1)$ and is therefore an important step towards optimal prover complexity.

Concurrent Work. Zhang et al. [ZGK+18] have concurrently with our work developed and implemented a scheme for verifying RAM computations. Like us and [ZGK+17], they avoid the use of permutation networks by using permutation proofs based on polynomial invariance by Neff [Nef01]. The idea underlying their technique for proving the correct fetch of an operation is related to the idea underpinning our look-up proofs. There are significant differences between the techniques used in our works; e.g. they rely on techniques from [CMT12] for instantiating proofs where we use techniques based on ideal linear commitments [BCG+17]. The proofs in [ZGK+18] are not zero-knowledge since they leak the number of times each type of instruction is executed, while our proofs are zero-knowledge. In terms of prover efficiency, [ZGK+18] focuses on concrete efficiency and yields impressive concrete performance. Asymptotically speaking, however, we are a polylogarithmic factor more efficient. This may require some explanation because they claim linear complexity for the prover. The reason is that they treat the prover as a TinyRAM machine with logarithmic word size in their performance measurement. Looking under the hood, we see that they use bit-decomposition to handle logical operations, which is constant overhead when

you fix a particular word size (e.g. 32 bits) but asymptotically the cost of this is logarithmic since it is linear in the word size. Also, they base commitments on cyclic groups and the use of exponentiations incurs a superlogarithmic overhead for the prover when implemented in TinyRAM.

Setup and Assumptions. Many proof systems, such as SNARKs, require a large and complex common reference string in order to run. The common reference string must be generated correctly, or the security of the proof system is at stake. This leads to concerns over parameter subversion, and efficiency, since the more complex the common reference string, the more costly it is to ensure that it was generated correctly. Recently, alternatives have been investigated. Hyrax [WTas+17] relies on the discrete logarithm assumption, and Ligero [AHIV17] and STARKs [BSBTHR18] rely on collision-resistant hash functions. Our scheme relies only on collision-resistant hash functions for soundness, and pseudorandom generators in order to achieve full zero-knowledge, which means that the setup information required is comparable to existing works, like STARKs, which focus on transparency.

Our proof system benefits from simple setup ingredients, nearly linear prover costs, and sublinear, hence, scalable communication and verification costs, and therefore enjoys many of the same desirable properties as STARKs [BSBTHR18].

In addition, although we do not know how to prove that our scheme is secure in any quantum security model, it is based on post-quantum assumptions and may offer some security against quantum adversaries, since it is not known how to efficiently attack collision-resistant hash functions and pseudorandom generators using quantum algorithms. Note that there are general proof systems, such as ZKB++ [CDG+17], which do have quantum proofs of security, but are asymptotically less efficient as previously discussed.

2 Preliminaries

2.1 Notation

We write $y \leftarrow A(x)$ for an algorithm returning y on input x. When the algorithm is randomized, we write $y \leftarrow A(x;r)$ to explicitly refer to the random coins r picked by the algorithm. We use a security parameter λ to indicate the desired level of security. The higher the security parameter, the smaller the risk of an adversary compromising security should be. For functions $f, g : \mathbb{N} \rightarrow [0,1]$, we write $f(\lambda) \approx g(\lambda)$ if $|f(\lambda) - g(\lambda)| = \frac{1}{\lambda^{\omega(1)}}$. We say a function f is *overwhelming* if $f(\lambda) \approx 1$ and that it is *negligible* if $f(\lambda) \approx 0$. In general we want the adversary's chance of breaking our proof systems to be negligible in λ. As a minimum requirement for an algorithm or adversary to be efficient it has to run in polynomial time in the security parameter. We abbreviate probabilistic (deterministic) polynomial time in the security parameter PPT (DPT). For a positive integer n, $[n]$ denotes the set $\{1, \ldots, n\}$. We use bold letters such as $\boldsymbol{v}$ for row vectors over a finite field $\mathbb{F}$.

2.2 Proofs of Knowledge

We follow [BCG+17] in defining proofs of knowledge over a communication channel and their specification of the ideal linear commitment channel and the standard channel. A *proof system* is defined by stateful PPT algorithms $(\mathcal{K}, \mathcal{P}, \mathcal{V})$. The setup generator $\mathcal{K}$ is only run once to provide public parameters pp that will be used by the prover $\mathcal{P}$ and verifier $\mathcal{V}$. We will in our security definitions just assume $\mathcal{K}$ is honest, which is reasonable since in our constructions the public parameters are publicly verifiable and could even be generated by the verifier.

The prover and verifier communicate with each other through a *communication channel* $\overset{\text{chan}}{\longleftrightarrow}$. When $\mathcal{P}$ and $\mathcal{V}$ interact on inputs s and t through a channel $\overset{\text{chan}}{\longleftrightarrow}$ we let $\mathsf{view}_{\mathcal{V}} \leftarrow \langle \mathcal{P}(s) \overset{\text{chan}}{\longleftrightarrow} \mathcal{V}(t) \rangle$ be the view of the verifier in the execution, i.e., all inputs he gets including random coins, and we let $\mathsf{trans}_{\mathcal{P}} \leftarrow \langle \mathcal{P}(s) \overset{\text{chan}}{\longleftrightarrow} \mathcal{V}(t) \rangle$ denote the transcript of the communication between prover and channel. The protocol ends with the verifier accepting or rejecting the proof. We write $\langle \mathcal{P}(s) \overset{\text{chan}}{\longleftrightarrow} \mathcal{V}(t) \rangle = b$ depending on whether he accepts $(b = 1)$ or rejects $(b = 0)$.

In the *standard channel* $\longleftrightarrow$, all messages are forwarded between prover and verifier. As in [BCG+17], we also consider an *ideal linear commitment* channel, $\overset{\mathsf{ILC}}{\longleftrightarrow}$, described in Fig. 2. When using the ILC channel, the prover can submit a `commit` command to commit to vectors of field elements of some fixed length k, specified in the public parameters. The vectors remain secretly stored in the channel, and will not be forwarded to the verifier. Instead, the verifier only learns how many vectors the prover has committed to. The verifier can submit a `send` command to the ILC channel to send a message to the prover. In addition, the verifier can also submit `open` queries to the ILC channel to obtain openings of linear combinations of the vectors sent by the prover. We stress that the verifier can request several linear combinations of stored vectors within a single `open` query, as depicted in Fig. 2 using matrix notation.

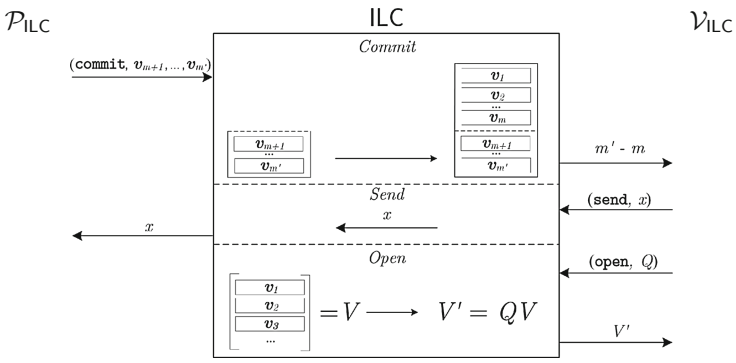

Fig. 2. Description of the ILC channel.

We say a proof system is *public coin* if the verifier's messages to the communication channel are chosen uniformly at random and independently of the actions of the prover, i.e., the verifier's messages to the prover correspond to the verifier's randomness ρ. All our proof systems will be public coin. In a proof system over the ILC channel, sequences of `commit`, `send` and `open` queries can alternate arbitrarily. However, since our proof systems are public coin we can without loss of generality assume the verifier will only make one big `open` query at the end of the protocol and then decide whether to accept or reject.

Let $\mathcal{R}$ be an efficiently decidable relation of tuples (pp, u, w). We can define a matching language $\mathcal{L} = \{(pp, u) | \exists w : (pp, u, w) \in \mathcal{R}\}$. We refer to u as the *instance* and w as the *witness* to $(pp, u) \in \mathcal{L}$. The public parameter pp will specify the security parameter λ, perhaps implicitly through its length, and may also contain other parameters used for specifying the relation. Typically, pp will also contain parameters that do not influence membership of $\mathcal{R}$ but may aid the prover and verifier, for instance the field and vector size in the ILC channel.

The protocol $(\mathcal{K}, \mathcal{P}, \mathcal{V})$ is called a *proof of knowledge* over a communication channel $\overset{\text{chan}}{\longleftrightarrow}$ for a relation $\mathcal{R}$ if it has perfect completeness and computational knowledge soundness as defined below.

Definition 1 (Perfect Completeness). *A proof system is* perfectly complete *if for all PPT adversaries* $\mathcal{A}$

$$\Pr\left[\begin{array}{c} pp \leftarrow \mathcal{K}(1^\lambda); (u, w) \leftarrow \mathcal{A}(pp) : \\ (pp, u, w) \notin \mathcal{R} \ \vee \ \langle \mathcal{P}(pp, u, w) \overset{chan}{\longleftrightarrow} \mathcal{V}(pp, u) \rangle = 1 \end{array}\right] = 1.$$

Definition 2 (Knowledge soundness). *A public-coin proof system has* computational (strong black-box) knowledge soundness *if for all DPT* $\mathcal{P}^*$ *there exists an expected PPT extractor* $\mathcal{E}$ *such that for all PPT adversaries* $\mathcal{A}$

$$\Pr\left[\begin{array}{c} pp \leftarrow \mathcal{K}(1^\lambda); (u, s) \leftarrow \mathcal{A}(pp); w \leftarrow \mathcal{E}^{\langle \mathcal{P}^*(s) \overset{chan}{\longleftrightarrow} \mathcal{V}(pp,u) \rangle}(pp, u) : \\ b = 1 \ \wedge \ (pp, u, w) \notin \mathcal{R} \end{array}\right] \approx 0.$$

Here the oracle $\langle \mathcal{P}^*(s) \overset{chan}{\longleftrightarrow} \mathcal{V}(pp, u) \rangle$ *runs a full protocol execution and if the proof is successful it returns the transcript* $\mathsf{trans}_{\mathcal{P}}$ *of the prover's communication with the channel. The extractor* $\mathcal{E}$ *can ask the oracle to rewind the proof to any point in a previous transcript and execute the proof again from this point on with fresh public-coin challenges from the verifier. We let* $b \in \{0, 1\}$ *be the verifier's output in the first oracle execution, i.e., whether it accepts or not, and we think of* s *as the state of the prover. The definition can then be paraphrased as saying that if the prover in state* s *makes a convincing proof, then* $\mathcal{E}$ *can extract a witness.*

If the definition holds also for unbounded $\mathcal{P}^*$ *and* $\mathcal{A}$ *we say the proof has* statistical knowledge soundness.

If the definition holds for a non-rewinding extractor, i.e., $\mathcal{E}$ *only requires a single transcript of the prover's communication with the channel, we say the proof system has knowledge soundness with* straight-line extraction.

We will construct public-coin proofs of knowledge that have special honest-verifier zero-knowledge. This means that if the verifier's challenges are known in advance then it is possible to simulate the verifier's view without knowing a witness. In our definition, the simulator works even for verifiers who may use adversarial biased coins in choosing their challenges as long as they honestly follow the specification of the protocol.

Definition 3 (Special Honest-Verifier Zero-Knowledge). *A public-coin proof of knowledge is* computationally special honest-verifier zero-knowledge (SHVZK) *if there exists a PPT simulator $\mathcal{S}$ such that for all stateful interactive PPT adversaries $\mathcal{A}$ that output randomness ρ for the verifier, and (u, w) such that $(pp, u, w) \in R$,*

$$\Pr\left[\begin{array}{c} pp \leftarrow \mathcal{K}(1^\lambda); (u, w, \rho) \leftarrow \mathcal{A}(pp); \\ \mathsf{view}_{\mathcal{V}} \leftarrow \langle \mathcal{P}(pp, u, w) \overset{chan}{\longleftrightarrow} \mathcal{V}(pp, u; \rho) \rangle : \mathcal{A}(\mathsf{view}_{\mathcal{V}}) = 1 \end{array}\right]$$
$$\approx \Pr\left[pp \leftarrow \mathcal{K}(1^\lambda); (u, w, \rho) \leftarrow \mathcal{A}(pp); \mathsf{view}_{\mathcal{V}} \leftarrow \mathcal{S}(pp, u, \rho) : \mathcal{A}(\mathsf{view}_{\mathcal{V}}) = 1\right].$$

We say the proof is statistically SHVZK *if the definition holds also against unbounded adversaries, and we say the proof is* perfectly SHVZK *if the probabilities are exactly equal.*

2.3 TinyRAM

TinyRAM is a random-access machine model operating on W-bit words and using K registers. We now describe the key features of TinyRAM but refer the reader to the specification [BSCG+13] for full details. A state of the TinyRAM machine consists of a program P (list of L instructions), a program counter pc (word), K registers $\mathsf{reg}_0, \ldots, \mathsf{reg}_{K-1}$ (words), a condition flag flag (bit), and M words of memory with addresses $0, \ldots, M-1$.

The TinyRAM specification includes two read-only tapes to retrieve its inputs but with little loss of efficiency we may assume the program starts by reading the tapes into memory[5] We will therefore skip the reading phase and assume the memory is initialized with the inputs (and 0 for the remaining words). Also, we will assume on initialization that pc, the registers and flag are all set to 0.

The program consists of a sequence of L instructions that include bit-wise logical operations, arithmetic operations, shifts, comparisons, jumps, and storing and loading data in memory. The program terminates by using a special command **answer** that returns a word. A description of the allowed operations is given in Table 1. We consider the program to have succeeded if it answers 0, otherwise we consider the answer to be a failure code.

We write reg_i and r_i when referring to register i and to its content, respectively. We write A to refer to either a register or an immediate value specified in a program instruction and write A for the value stored therein. Depending on the

[5] The specification [BSCG+13] calls a program *proper* if it first reads all inputs into memory and provides a 7-line TinyRAM program that does this in $\sim 5M$ steps.

instruction a word a may be interpreted as an unsigned value in $\{0, \ldots, 2^W - 1\}$ or as a signed value in $\{-2^{W-1}, \ldots, 2^{W-1} - 1\}$. Signed values are in two's complement, so given a word $\mathsf{a} = (a_{w-1}, \ldots, a_0) \in \{0,1\}^W$ the bit a_{W-1} is the sign and the signed value is $-2^W + \mathsf{a}$ if $a_{W-1} = 1$ and a if $a_{W-1} = 0$. We distinguish operations over signed values by using subscript s, e.g. $\mathsf{a} \times_s \mathsf{b}$ and $\mathsf{a} \geq_s \mathsf{b}$ are used to denote product and comparison over the signed values.

Correct Program Execution. It is often important to check that a protocol participant supposedly running program P on public input x and private input w provides the correct output z. Without loss of generality, we can formulate the verification as an extended program that takes public input $v = (x, z)$ and answers 0 if and only if z is the output of the computation. We therefore formulate correct program execution as the program just answering 0.

We now give a relation that captures correct TinyRAM program execution. An instance is of the form $u = (P, v, T, M)$, where P is a TinyRAM program, v is a list of words given as input to the program, T is a time bound, and M is the size of the memory. A witness w is another list of words. We assume without loss of generality that the witness is appended by 0's, such that $|v| + |w| = M$ and the program starts with the memory being initialized to these words.

The statement we want to prove is that the program P terminates in T steps using M words of memory on the public input v and private input w with the instruction **answer** 0. We therefore define

$$\mathcal{R}_{\mathsf{TinyRAM}} = \left\{ \begin{array}{c} (pp, u, w) = ((W, K, *), (P, v, T, M), w) \ \mid \\ P \text{ is a TinyRAM program with } W\text{-bit words, } K \text{ registers,} \\ \text{and } M \text{ words of addressable memory, which on inputs } v \text{ and } w \\ \text{terminates in } T \text{ steps with the instruction } \textbf{answer } 0. \end{array} \right\}$$

Our main interest is to prove correct execution of programs that require heavy computation so we will throughout the article assume the number of steps outweigh the other parameters, i.e., $T > L + M$, where L is the number of instructions in the program.

3 Arithmetization of Correct Program Execution

As a first step towards the realization of proofs for the correct execution of TinyRAM programs we translate $\mathcal{R}_{\mathsf{TinyRAM}}$ into a more amenable relation involving elements in a finite field. Given a TinyRAM machine with word-size W and a finite field $\mathbb{F}$, we can in a natural way embed words into field elements by encoding a word $a \in \{0, \ldots, 2^W - 1\}$ as the field element $a \cdot 1_{\mathbb{F}} = 1_{\mathbb{F}} + \cdots + 1_{\mathbb{F}}$ (a times). We will use fields of characteristic $p > 2^{2W} - 2^{W-1}$ because then sums and products of words are less than p and we avoid overflow in the field operations we apply to the embedded words.

We will encode the program, memory and states of a TinyRAM program as tuples of field elements. We then introduce a new relation $\mathcal{R}^{\mathsf{field}}_{\mathsf{TinyRAM}}$ consisting of a set of arithmetic constraints these encodings should satisfy to guarantee

the correct program execution. The relation will take instances $u = (P, v, T, M)$, and witnesses w consisting of the encodings as well as a set of auxiliary field elements.

In this section we specify instructions supported by TinyRAM machines and the structure of the witness w and how the relation of correct program execution decomposes into simpler sub-relations. It will be the case that the encoding of the witness can be done alongside an execution of the program in $\mathcal{O}(L+M+T)$ field operations.

Table 1 described the supported operations in TinyRAM. Each line in the program consists of one of these instructions in and up to three operands, e.g. **add** reg_i reg_j A. The first operand, reg_i, usually points to the register storing the result of the operation, **add**, computed on the words specified by the next two operands, reg_j, A. The last operand A indicates an immediate value that could be either used directly in the operation or to point to the content of another register. We refer to the value to be used in the operation generically as A, stressing that the selection between either the immediate value or a register value can be handled by using the appropriate selection vector.

Formatting the Witness. Given a correct program execution we encode program, memory and states of the TinyRAM machine as field elements and arrange them in a number of tables as pictured in Table 2. The execution table Exe, contains the field elements encoding of the states of the TinyRAM machine. It consists of T rows, where row t describes the state at the beginning of step t. A row includes field elements that encode the time t, the program counter pc_t, the instruction $\mathsf{inst}_{\mathsf{pc}_t}$ corresponding to pc_t, an immediate value A_t, the values $\mathsf{r}_{0,t}, \ldots, \mathsf{r}_{K-1,t}$ contained in the registers $\mathsf{reg}_0, \ldots, \mathsf{reg}_{K-1}$ at time t, and the flag flag_t. The next row contains the resulting state of the TinyRAM machine at time $t+1$. Each row also includes a memory address addr_t, and the value $\mathsf{v}_{\mathsf{addr}_t}$ stored at this address after the execution of the step, as well as a constant number of auxiliary field elements to be specified later that will be used to check correctness of program execution.

The next table is the program table Prog, which contains the field elements encoding of the TinyRAM program P. Each row contains the description of one line of the program, consisting of one instruction, at most three operands, and possibly an immediate value. Furthermore, we introduce a constant number of auxiliary field elements in each row. These entries can be efficiently computed given the program line stored in the same row and will help verifying its execution, e.g. we encode the position of input and output registers as auxiliary field elements.

The memory table Mem has rows that list the possible memory addresses, their initial values, and an auxiliary field element $\mathsf{usd} \in \{0, 1\}$. For every pair of address and corresponding initial value, the memory table Mem contains a row in which $\mathsf{usd} = 0$ and another row in which $\mathsf{usd} = 1$. Recall that the memory is initialized with input words listed in v, w, i.e., the input words contributing to the instance and witness of the relation $\mathcal{R}_{\mathsf{TinyRAM}}$.

Table 1. TinyRAM instruction set, excluding the **read** command. The flag is set equal to 1 if the condition is met and 0 otherwise. If the pc exceeds the program length, i.e., $\mathsf{pc} \geq L$, or we address a non-existing part of memory, i.e., in a **store** or **load** instruction $\mathsf{A} \geq M$, the TinyRAM machine halts with answer 1.

Instruction	Operands	Effect	Flag
and	reg_i reg_j A	Compute r_i as bitwise AND of r_j and A	Result is 0^W
or	reg_i reg_j A	Compute r_i as bitwise OR of r_j and A	Result is 0^W
xor	reg_i reg_j A	Compute r_i as bitwise XOR of r_j and A	Result is 0^W
not	reg_i A	Compute r_i as bitwise NOT of A	Result is 0^W
add	reg_i reg_j A	Compute $\mathsf{r}_i = \mathsf{r}_j + \mathsf{A} \bmod 2^W$	Overflow: $\mathsf{r}_j + \mathsf{A} \geq 2^W$
sub	reg_i reg_j A	Compute $\mathsf{r}_i = \mathsf{r}_j - \mathsf{A} \bmod 2^W$	Borrow: $\mathsf{r}_j < \mathsf{A}$
mull	reg_i reg_j A	Compute $\mathsf{r}_i = \mathsf{r}_j \times \mathsf{A} \bmod 2^W$	$\neg$ overflow: $\mathsf{r}_j \times \mathsf{A} < 2^W$
umulh	reg_i reg_j A	Compute r_i as upper W bits of $\mathsf{r}_j \times \mathsf{A}$	$\neg$ overflow: $\mathsf{r}_i = 0$
smulh	reg_i reg_j A	Compute r_i as upper W bits of the signed $2W$-bit $\mathsf{r}_j \times_s \mathsf{A}$ (**mull** gives lower word)	$\neg$ over/underflow: $\mathsf{r}_i = 0$
udiv	reg_i reg_j A	Compute r_i as quotient of r_j/A	Division by zero: $\mathsf{A} = 0$
umod	reg_i reg_j A	Compute r_i as remainder of r_j/A	Division by zero: $\mathsf{A} = 0$
shl	reg_i reg_j A	Compute r_i as r_i shifted left by A bits	MSB of r_j
shr	reg_i reg_j A	Compute r_i as r_i shifted right by A bits	LSB of r_j
cmpe	reg_i A	Compare if equal	Equal: $\mathsf{r}_i = \mathsf{A}$
cmpa	reg_i A	Compare if above	Above: $\mathsf{r}_i > \mathsf{A}$
cmpae	reg_i A	Compare if above or equal	Above/equal: $\mathsf{r}_i \geq \mathsf{A}$
cmpg	reg_i A	Signed compare if greater	Greater: $\mathsf{r}_i >_s \mathsf{A}$
cmpge	reg_i A	Signed compare if greater or equal	Greater/equal: $\mathsf{r}_i \geq_s \mathsf{A}$
mov	reg_i A	Set $\mathsf{r}_i = \mathsf{A}$	Flag unchanged
cmov	reg_i A	if $\mathsf{flag} = 1$ set $\mathsf{r}_i = \mathsf{A}$	Flag unchanged
jmp	A	Set $\mathsf{pc} = A$	Flag unchanged
cjmp	A	If $\mathsf{flag} = 1$ set $\mathsf{pc} = \mathsf{A}$	Flag unchanged
cnjmp	A	If $\mathsf{flag} = 0$ set $\mathsf{pc} = \mathsf{A}$	Flag unchanged
store	A reg_i	Store in memory address A the word r_i	Flag unchanged
load	reg_i A	Set r_i to the word stored at address A	Flag unchanged
answer	A	Stall or halt returning the word A	Flag unchanged

Table 2. The execution table Exe, the program table Prog, the memory table Mem and the table EvenBits.

Time	pc	Instruction	Immediate	reg_0	...	reg_{K-1}	Flag	Address	Value	$\boldsymbol{aux}_{\mathsf{Exe}}$
1	0	inst_0	A_0	0	...	0	0	0	0	...
					⋮					
t	pc_t	$\mathsf{inst}_{\mathsf{pc}_t}$	A_t	$\mathsf{r}_{0,t}$	...	$\mathsf{r}_{K-1,t}$	flag_t	addr_t	$\mathsf{v}_{\mathsf{addr}_t}$	...
$t+1$	pc_{t+1}	$\mathsf{inst}_{\mathsf{pc}_{t+1}}$	A_{t+1}	$\mathsf{r}_{0,t+1}$	...	$\mathsf{r}_{K-1,t+1}$	flag_{t+1}	addr_{t+1}	$\mathsf{v}_{\mathsf{addr}_{t+1}}$	...
					⋮					
T	pc_T	**answer** 0	0	$\mathsf{r}_{0,T}$	...	$\mathsf{r}_{K-1,T}$	flag_T	addr_T	$\mathsf{v}_{\mathsf{addr}_T}$	...

(a) The execution table Exe.

pc	Instruction	Immediate	$\boldsymbol{aux}_{\mathsf{Prog}}$
0	inst_0	A_0	...
	⋮		
$L-1$	inst_{L-1}	A_{L-1}	...

(b) The program table Prog.

Address	Initial value	usd
0	0	0
1	v_1	0
	⋮	
$M-1$	v_{M-1}	0
0	0	1
1	v_1	1
	⋮	
$M-1$	v_{M-1}	1

(c) The memory table Mem.

Values
0
1
4
5
⋮
$\sum_{i=0}^{\frac{W}{2}-1} 2^{2i}$

(d) The table EvenBits.

In addition to these, we also consider an auxiliary lookup table EvenBits containing the encoding of words of length W whose binary expansion has 0 in all odd positions. The table contains $2^{\frac{W}{2}}$ field elements and will be used as part of a check that certain field elements encode a word of length W.

3.1 Decomposition of TinyRAM

Let $(\mathsf{Exe}, \mathsf{Prog}, \mathsf{Mem}, \mathsf{EvenBits})$ be the tables of field elements encoding the program execution and the auxiliary values. We can now reformulate the correct execution of a TinyRAM program defined by $\mathcal{R}_{\mathsf{TinyRAM}}$ as a relation that imposes a number of constraints the field elements included in tables should satisfy:

$$\mathcal{R}^{\mathsf{field}}_{\mathsf{TinyRAM}} = \left\{ \begin{array}{c} (pp, u, w) = ((W, K, \mathbb{F}, *), (P, v, T, M), \mathsf{w}) \;\Big|\; \\ \mathsf{w} = (\mathsf{Exe}, \mathsf{Prog}, \mathsf{Mem}, \mathsf{EvenBits}, *) \\ (pp, (P, v, T, M), \mathsf{w}) \in \mathcal{R}_{\mathsf{check}} \\ (pp, (T, M), \mathsf{w}) \in \mathcal{R}_{\mathsf{mem}} \\ (pp, \perp, \mathsf{w}) \in \mathcal{R}_{\mathsf{step}} \end{array} \right\}$$

where the relations $\mathcal{R}_{\mathsf{check}}$, $\mathcal{R}_{\mathsf{mem}}, \mathcal{R}_{\mathsf{step}}$ jointly guarantee the witness w consists of field elements encoding a correct TinyRAM execution that answers 0 in T steps using M words of memory, public input v, and additional private inputs.

Specifically, the relation $\mathcal{R}_{\mathsf{check}}$ checks the initial values of the memory are correctly included into Mem, the program is correctly encoded in Prog, EvenBits

contains the correct encodings of the auxiliary lookup table, the initial state of the TinyRAM machine is correct and that it terminates with answer 0 in step T. The role of $\mathcal{R}_{\mathsf{mem}}$ is to check that memory usage is consistent throughout the execution of the program. That is, if a memory value is loaded at time t then it should match the last stored value at the same address. Finally, $\mathcal{R}_{\mathsf{step}}$ checks that each step of the execution has been performed correctly. In the rest of the section we describe $\mathcal{R}_{\mathsf{check}}$, $\mathcal{R}_{\mathsf{mem}}$ and $\mathcal{R}_{\mathsf{step}}$, gradually decomposing them into smaller and simpler relations. Ultimately, we specify each of these subrelations in terms of some building block: equality, lookup, permutation, and range relations. Figure 3 illustrates the decomposition of $\mathcal{R}^{\mathsf{field}}_{\mathsf{TinyRAM}}$ into progressively smaller relations.

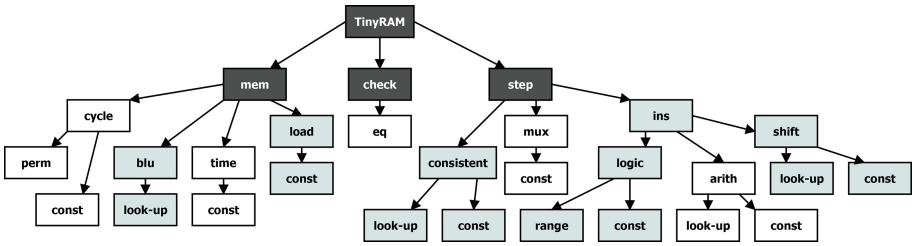

Fig. 3. Diagram of the decomposition of TinyRAM into equality, lookup, permutation, and range relations.

Building Blocks. We give a brief description of the building block relations used in the decomposition of $\mathcal{R}^{\mathsf{field}}_{\mathsf{TinyRAM}}$.

- An equality relation $\mathcal{R}_{\mathsf{eq}}$ checks that rows Tab_i of a table Tab in the witness encode tuples $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m$ of given W-bit words
- A lookup relation checks the membership of a tuple of field elements $\boldsymbol{w}$ in the set of rows of a table Tab. This differs from the previous relation as both $\boldsymbol{w}$ and Tab are both in the witness. We extend this relation in the natural way for checking the membership of multiple tuples $\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots$ in a table.
- A range relation to check that a field element a can be written as a W-bit word, i.e., a is in the range $\{0, \ldots, 2^W - 1\}$.
- A permutation relation can be used to check that two ordered sets of a given size are permutations of each other. The permutation is in the witness i.e. it is unknown to the verifier.

3.2 Checking the Correctness of Values

The role of $\mathcal{R}_{\mathsf{check}}$ is to check that w consists of the correct number of field elements that can be partitioned into the appropriate tables and also to check that specific entries in these tables are correct.

$$\mathcal{R}_{\mathsf{check}} = \left\{ \begin{array}{l} (pp, u, w) = ((W, K, \mathbb{F}, *), (P, v, T, M), \mathsf{w}) \;\big|\; \\ \mathsf{w} = (\mathsf{Exe}, \mathsf{Prog}, \mathsf{Mem}, \mathsf{EvenBits}, *), \\ \mathsf{Exe} = \{\mathsf{Exe}_t\}_{t=1}^{T}, \quad \mathsf{Prog} = \{\mathsf{Prog}_i\}_{i=0}^{L-1} \\ \mathsf{Prog}_0 = (0, \mathsf{inst}_0, \mathsf{A}_0, \ldots) \\ (pp, (1, 0, \mathsf{inst}_0, A_0, 0, \ldots, 0, \ldots), \mathsf{Exe}_1) \in \mathcal{R}_{\mathsf{eq}} \\ (pp, (T, \mathbf{answer}, 0, \ldots), \mathsf{Exe}_T) \in \mathcal{R}_{\mathsf{eq}} \\ \left(pp, \left(0, 1, 4, 5, \ldots, \sum_{i=0}^{\frac{W}{2}-1} 2^{2i}\right), \mathsf{EvenBits}\right) \in \mathcal{R}_{\mathsf{eq}} \\ (pp, P, \mathsf{Prog}) \in \mathcal{R}_{\mathsf{eq}} \quad (pp, v, \mathsf{Mem}) \in \mathcal{R}_{\mathsf{eq}} \end{array} \right\}.$$

The relation $\mathcal{R}_{\mathsf{check}}$ checks that: the first and last row of the execution table contains the correct initial values; the auxiliary lookup table $\mathsf{EvenBits}$ contains the embeddings of all W-bit words with 0 in all odd positions; the program table Prog contains the correct field element embedding of the program P as well as the correct auxiliary entries; the memory table Mem contains the correct embedding of the input words listed in v.

3.3 Checking Memory Consistency

The relation $\mathcal{R}_{\mathsf{mem}}$ checks that the memory is used consistently across different steps in the execution. For instance, if at step t a value is loaded from memory, then it should be equal to the last value stored in the same address. If it is the first time a memory address is accessed, we need to ensure consistency with the initial values. If two consecutive memory accesses to the same address were placed into two adjacent rows of Exe it would be easy to check their consistency. However, this is generally not the case since the Exe table is sorted by execution time rather than memory access. Therefore, we need to devise a way to check consistency of memory accesses that could be located in any position of Exe. Overall the memory consistency relation $\mathcal{R}_{\mathsf{mem}}$ decomposes as follows

$$\mathcal{R}_{\mathsf{mem}} = \left\{ \begin{array}{ll} (pp, u, w) = ((W, K, \mathbb{F}, *), (T, M), \mathsf{w}) \;\big|\; & \\ \mathsf{w} = (\mathsf{Exe}, \mathsf{Prog}, \mathsf{Mem}, \mathsf{EvenBits}, \pi, *), & \\ \mathsf{Exe} = \{\mathsf{Exe}_t\}_{t=1}^{T} & \mathsf{Mem} = \{\mathsf{Mem}_j\}_{j=0}^{2M-2} \\ (pp, T, (\mathsf{Exe}, \pi)) \in \mathcal{R}_{\mathsf{cycle}}, & (pp, T, \mathsf{Exe}) \in \mathcal{R}_{\mathsf{time}} \\ (pp, (T, M), (\mathsf{Exe}, \mathsf{Mem})) \in \mathcal{R}_{\mathsf{blookup}}, & (pp, T, \mathsf{Exe}) \in \mathcal{R}_{\mathsf{load}} \end{array} \right\}$$

To help with checking the memory consistency, we include in each row of the execution table the following auxiliary entries

$$\boldsymbol{aux}_{\mathsf{Exe}} = \boxed{\tau_{\mathrm{link}}} \boxed{\mathsf{v}_{\mathrm{link}}} \boxed{\mathsf{v}_{\mathrm{init}}} \boxed{\mathsf{usd}} \boxed{\mathsf{S}} \boxed{\mathsf{L}} \boxed{\cdots}$$

where τ_{link} contains the previous time-step at which the current address was accessed, unless this is the first time a location is accessed in which case it is set equal to the last time-step this location is accessed. Similarly, $\mathsf{v}_{\mathrm{link}}$ stores the value contained in the address after time τ_{link}, unless this is the first time that

location is accessed, in which case it stores the last value stored in that location. The value v_{init} is a copy of the initial value assigned to that memory location, which is also stored in the memory table Mem. The value usd is a flag which is set equal to 0 if this is the first time we access the current memory address, and 1 otherwise. The values S, L are flags set equal to 1 in case the current instruction is a **store** or **load** operation, respectively, and 0 otherwise. The values S, L are also stored in the auxiliary entries of the program table $\boldsymbol{aux}_{\mathsf{Prog}} = \boxed{\mathsf{S}}\boxed{\mathsf{L}}\boxed{\cdots}$.

Memory Accesses Form Cycles. We check memory consistency by specifying cycles of memory accesses, so that consecutive terms in a cycle correspond to two consecutive accesses to the same memory location. By using the above auxiliary entries, we use the relation $\mathcal{R}_{\mathsf{cycle}}$ for the memory access pattern in the rows of Exe being in correspondence with a permutation π defined by such cycles. The relation $\mathcal{R}_{\mathsf{cycle}}$ checks that all memory accesses (i.e. with $\mathsf{S} + \mathsf{L} = 1$) relative to the same address are connected into cycles and that rows not involving memory operations ($\mathsf{S} + \mathsf{L} = 0$) are not included in these cycles. The relation does not include any explicit checks on whether $\mathsf{S} + \mathsf{L}$ is equal to 0 or 1. It is sufficient to check that $\mathsf{S}_t + \mathsf{L}_t = \mathsf{S}_{t'} + \mathsf{L}_{t'}$, $t = \tau_{\mathrm{link}t'}$, $\mathsf{v}_{\mathsf{addr}_t} = \mathsf{v}_{\mathrm{link}t'}$ and $\mathsf{addr}_t = \mathsf{addr}_{t'}$ for some $t' = \pi(t)$, which ensures that operations which are not memory operations are not part of cycles including memory operations.

$$\mathcal{R}_{\mathsf{cycle}} = \left\{ \begin{array}{c} (pp, u, w) = ((W, K, \mathbb{F}, *), T, (\mathsf{Exe}, \pi)) \ \big| \\ \mathsf{Exe}_t = (t, \ldots, \mathsf{addr}_t, \mathsf{v}_{\mathrm{link}t}, \tau_{\mathrm{link}t}, \ldots, S_t, L_t, \ldots) \text{ for } t \in [T] \\ \boldsymbol{a}_t = (t, \mathsf{addr}_t, \mathsf{v}_{\mathsf{addr}_t}, \mathsf{S}_t + \mathsf{L}_t) \text{ for } t \in [T] \\ \boldsymbol{b}_t = (\tau_{\mathrm{link}t}, \mathsf{addr}_t, \mathsf{v}_{\mathrm{link}t}, \mathsf{S}_t + \mathsf{L}_t) \text{ for } t \in [T] \\ ((W, K, \mathbb{F}, *), T, (\{\boldsymbol{a}_i, \boldsymbol{b}_i\}_{i=1}^{T}, \pi)) \in \mathcal{R}_{\mathsf{perm}} \end{array} \right\}$$

Memory Accesses are in the Correct Order. Consecutive terms in a cycle should correspond to the consecutive time-steps in which the memory is accessed. To check that the memory cycles are time-ordered we can simply verify that $t > \tau_{\mathrm{link}t}$ for any given time-step $t \in [T]$[6]. Since memory accesses are connected into cycles, the first time we access a new memory location the τ_{link} entry stores the last point in time that location is accessed by the program. In this case ($\mathsf{usd} = 0$), we verify that $t \leq \tau_{\mathrm{link}t}$. The relation $\mathcal{R}_{\mathsf{time}}$ incorporates these conditions

$$\mathcal{R}_{\mathsf{time}} = \left\{ \begin{array}{c} (pp, u, w) = ((W, K, \mathbb{F}, *), T, \mathsf{Exe}) \ \big| \\ \mathsf{Exe}_t = (t, \ldots, \tau_{\mathrm{link}t}, \ldots, \mathsf{usd}_t, \ldots) \text{ for } t \in [T] \\ \forall\ t \in [T] : (\mathsf{usd} = 0 \wedge t \leq \tau_{\mathrm{link}t}) \vee (\mathsf{usd} = 1 \wedge t > \tau_{\mathrm{link}t}) \end{array} \right\}$$

Memory Locations are in no more than one Cycle. To ensure that the cycles correspond to sequences of memory addresses we also require that all the

[6] For this to be sufficient we also need the time-steps in the execution table to be correct but this is ensured by the $\mathcal{R}_{\mathsf{check}}$ and $\mathcal{R}_{\mathsf{consistent}}$ (appears later) relations.

rows touching the same memory address are included in the *same* cycle. Since the cycles are time-ordered, they require one time-step for which $\mathsf{usd} = 0$ in order to close a cycle. Thus, we can ensure each memory location to be part of at most on cycle by letting usd to be set equal to 0 at most once for each memory address. We introduce a *bounded* lookup relation $\mathcal{R}_{\mathsf{blookup}}$ to address this requirement. The relation checks that for any row in Exe, the tuple $(\mathsf{addr}_t, \mathsf{v}_{\mathrm{init}\,t}, \mathsf{usd})$ is contained in one row of the table Mem and that each row $(j, \mathsf{v}_j, 0)$ of Mem is accessed at most once by the program.

$$\mathcal{R}_{\mathsf{blookup}} = \left\{ \begin{array}{c} (pp, u, w) = ((W, K, \mathbb{F}, *), (T, M), (\mathsf{Exe}, \mathsf{Mem})) \ \big| \\ \mathsf{Exe}_t = (t, \ldots, \mathsf{addr}_t, \ldots, \mathsf{v}_{\mathrm{init}\,t}, \mathsf{usd}_t, \ldots) \text{ for } t \in [T] \\ \forall\ t \in [T]\ (pp, \bot, ((\mathsf{addr}_t, \mathsf{v}_{\mathrm{init}\,t}, \mathsf{usd}_t), \mathsf{Mem})) \in \mathcal{R}_{\mathsf{lookup}} \wedge \\ \forall\ \ (j, \mathsf{v}_j, 0) \in \mathsf{Mem} : (\ldots, j, \ldots, \mathsf{v}_j, 0, \ldots) \text{ occurs at most once in } \mathsf{Exe} \end{array} \right\}$$

Load Instructions are Consistent. Finally, we are only left to check that if the program executes a **load** instruction the value $\mathsf{v}_{\mathsf{addr}_t}$ loaded from memory is consistent with the value stored at the same address at the previous access. Similarly, if **load** is executed on a new memory location, then the value loaded should match with the initial value $\mathsf{v}_{\mathrm{init}\,t}$. No additional checks are required for **store** instructions. These checks are incorporated in the relation $\mathcal{R}_{\mathsf{load}}$.

$$\mathcal{R}_{\mathsf{load}} = \left\{ \begin{array}{c} (pp, u, w) = ((W, K, \mathbb{F}, *), T, \mathsf{Exe}) \ \big| \\ \mathsf{Exe}_t = (t, \ldots, \mathsf{addr}_t, \mathsf{v}_{\mathsf{addr}_t}, \tau_{\mathrm{link}\,t}, \mathsf{v}_{\mathrm{link}\,t}, \mathsf{v}_{\mathrm{init}\,t}, \mathsf{usd}_t, \ldots) \text{ for } t \in [T] \\ \forall\ t \in [T] : \mathsf{L}_t(\mathsf{v}_{\mathsf{addr}_t} - \mathsf{v}_{\mathrm{init}\,t} + \mathsf{usd}_t(\mathsf{v}_{\mathrm{init}\,t} - \mathsf{v}_{\mathrm{link}\,t})) = 0 \end{array} \right\}$$

3.4 Checking Correct Execution of Instructions

We use the relation $\mathcal{R}_{\mathsf{step}}$ to guarantee that each step of the execution has been performed correctly. This involves checking for each row Exe_t of the execution table that the stored words are in the range $\{0, \ldots, 2^W - 1\}$, the flag_t is a bit, the program counter pc_t matches the instruction and the immediate value A_t in the program, and that inst_t is correctly executed. An instruction takes some inputs, e.g., values indicated by the operands reg_j, A or the flag and as a result may change the program counter, a register value, a value stored at a memory address, or the flag. Since we have already checked memory correctness, if the operation is a load or store we may assume the memory value is correct.

$$\mathcal{R}_{\mathsf{step}} = \left\{ \begin{array}{l} (pp, u, w) = ((W, K, \mathbb{F}, *), \bot, \mathsf{w}) \ \big| \\ \mathsf{w} = (\mathsf{Exe}, \mathsf{Prog}, \mathsf{Mem}, \mathsf{EvenBits}, *) \ \wedge \ \mathsf{Exe} = \{\mathsf{Exe}_t\}_{t=1}^{T} \\ \forall t \in \{1, \ldots, T-1\} : \\ \qquad (pp, \bot, (\mathsf{Exe}_t, \mathsf{Exe}_{t+1})) \in \mathcal{R}_{\mathsf{mux}} \\ \qquad (pp, \bot, (\mathsf{Exe}_i, \mathsf{Exe}_{i+1}, \mathsf{Prog})) \in \mathcal{R}_{\mathsf{consistent}} \\ \qquad (pp, \bot, (\mathsf{Exe}_i, \mathsf{Exe}_{i+1}, \mathsf{EvenBits}, *)) \in \mathcal{R}_{\mathsf{ins}} \end{array} \right\}.$$

To help checking the consistency of the operations the rows of the execution and program tables include some auxiliary entries. These consist of some *temporary variables*, an output vector, and some *selection vectors* which are also listed in the program table. The temporary variables are used to store a copy of the inputs and outputs of an instruction. The advantage of the temporary variables is that for each addition operation we check, we will always have the inputs and outputs stored, instead of having to handle multiple registers holding inputs and output in arbitrary order.

Ensuring Temporary Values are Correct. A multiplexing relation $\mathcal{R}_{\mathsf{mux}}$ is used to check that the temporary variables are consistent with operands contained in inst_t. Checking operations on temporary values require us to multiplex the corresponding register, immediate, and memory values in and out of the temporary values. We do this using selection vectors that are bit-vectors encoding the operands of an instruction. Each row of the execution table includes multiple variables that may be selected as an operand. A selection vector will have a bit for each of these variables indicating whether it is picked or not. More details about the multiplexing relation are provided in the full version of the paper [BCG+18].

The Execution Table and the Program Table are Consistent. The consistency relation $\mathcal{R}_{\mathsf{consistent}}$ checks that the time is correctly incremented and that the program counter is in the correct range, i.e. $\mathsf{pc}_{t+1} \in \{0, \ldots, L-1\}$ and is incremented unless a jump-instruction is executed. It also checks that the instruction, the immediate value and the selection vectors stored in the execution table are consistent with the program the line indexed pc. Furthermore, it checks that the entries in the output vector relevant to inst_t are all equal to zero and that the contents of the registers do not change, unless specified by the instruction, e.g. the register storing the result of the computation. Verifying that rows of the execution table match with states of a TinyRAM machine involves checking that entries that are not affected by an instruction remain the same in the next state. For this we use another selector vector with entries equal to 0, positioned in correspondence of entries that are changed during the execution, and 1 for entries that do not change in the execution.

Instructions are Executed Correctly. An instruction checker relation $\mathcal{R}_{\mathsf{ins}}$ checking that the temporary values are in the range $\{0, \ldots, 2^W - 1\}$ and are consistent with the output vector. We divide the entries of the output vector into 4 groups: logical ($\mathsf{AND}, \mathsf{XOR}, \mathsf{OR}$), arithmetic ($\mathsf{SUM}, \mathsf{PROD}, \mathsf{SSUM}, \mathsf{SPROD}, \mathsf{MOD}$), shift ($\mathsf{SHIFT}$), and flag ($\mathsf{FLAG}_1, \mathsf{FLAG}_2, \mathsf{FLAG}_3, \mathsf{FLAG}_4$). By specifying constraints to all these entries, we can directly verify all the logical, arithmetic, and shifts operations after which the variables are named.

The $\mathcal{R}_{\mathsf{ins}}$ can be decomposed into 3 sub-relations: $\mathcal{R}_{\mathsf{logic}}$, $\mathcal{R}_{\mathsf{arith}}$, and $\mathcal{R}_{\mathsf{shift}}$. In the full paper [BCG+18] we show choices of selection vectors which reduce the verification of any other operation to the ones contained in these 3 categories. We also describe the decomposition of $\mathcal{R}_{\mathsf{logic}}$, $\mathcal{R}_{\mathsf{arith}}$, $\mathcal{R}_{\mathsf{shift}}$ into our elementary building blocks.

4 Efficient Bit Decomposition for Logical Relations

In this section we summarise a new decomposition technique which will enable verification of bitwise AND and XOR operations. This allows us to check all boolean operations more efficiently. Let a, b be the inputs of the bit-wise AND or bit-wise XOR operation, and let c be the output. To verify the correctness of the operation, e.g. $\mathsf{a} \wedge \mathsf{b} = \mathsf{c}$, consider the decompositions of the inputs into their odd and even-position bits, namely $\mathsf{a} = 2\mathsf{a}_\mathrm{o} + \mathsf{a}_\mathrm{e}$ and $\mathsf{b} = 2\mathsf{b}_\mathrm{o} + \mathsf{b}_\mathrm{e}$. Observe that taking the sum of the integers storing the even-positions of a and b gives

$$\begin{aligned}\mathsf{a}_\mathrm{e} + \mathsf{b}_\mathrm{e} &= (0, \mathsf{a}_{W-2}, \ldots, 0, \mathsf{a}_0) + (0, \mathsf{b}_{W-2}, \ldots, 0, \mathsf{b}_0) \\ &= (\mathsf{a}_{W-2} \wedge \mathsf{b}_{W-2}, \mathsf{a}_{W-2} \oplus \mathsf{b}_{W-2}, \ldots, \mathsf{a}_0 \wedge \mathsf{b}_0, \mathsf{a}_0 \oplus \mathsf{b}_0)\end{aligned}$$

The above contains the bit-wise AND of the even bits of a and b placed in odd position and the bit-wise XOR of the even bits of a and b in even position. Therefore we can consider taking again the decomposition of $\mathsf{a}_\mathrm{e} + \mathsf{b}_\mathrm{e}$ into its odd and even-position bits, i.e. $\mathsf{a}_\mathrm{e} + \mathsf{b}_\mathrm{e} = 2\mathrm{e}_\mathrm{o} + \mathrm{e}_\mathrm{e}$ so that half of the bits of $\mathsf{a} \wedge \mathsf{b}$ are stored in e_o and half of the bits of $\mathsf{a} \oplus \mathsf{b}$ are stored in e_e. We can repeat the above procedure starting from the odd-position bits of a and b getting the following

$$\begin{aligned}\mathsf{a}_\mathrm{o} + \mathsf{b}_\mathrm{o} &= (0, \mathsf{a}_{W-1}, \ldots, 0, \mathsf{a}_1) + (0, \mathsf{b}_{W-1}, \ldots, 0, \mathsf{b}_1) \\ &= (\mathsf{a}_{W-1} \wedge \mathsf{b}_{W-1}, \mathsf{a}_{W-1} \oplus \mathsf{b}_{W-1}, \ldots, \mathsf{a}_1 \wedge \mathsf{b}_1, \mathsf{a}_1 \oplus \mathsf{b}_1) = 2\mathrm{o}_\mathrm{o} + \mathrm{o}_\mathrm{e}\end{aligned}$$

where o_o stores half of the bits of $\mathsf{a} \wedge \mathsf{b}$ and o_e stores and half of the bits of $\mathsf{a} \oplus \mathsf{b}$. Putting everything together, given the decompositions $\mathsf{a}_\mathrm{o}, \mathsf{a}_\mathrm{e}, \mathsf{b}_\mathrm{o}, \mathsf{b}_\mathrm{e}, \mathrm{o}_\mathrm{o}, \mathrm{o}_\mathrm{e}, \mathrm{e}_\mathrm{o}, \mathrm{e}_\mathrm{e} \in \mathsf{EvenBits}$ such that the following hold

$$\mathsf{a} = 2\mathsf{a}_\mathrm{o} + \mathsf{a}_\mathrm{e} \qquad \mathsf{b} = 2\mathsf{b}_\mathrm{o} + \mathsf{b}_\mathrm{e} \qquad \mathsf{a}_\mathrm{o} + \mathsf{b}_\mathrm{o} = 2\mathrm{o}_\mathrm{o} + \mathrm{o}_\mathrm{e} \qquad \mathsf{a}_\mathrm{e} + \mathsf{b}_\mathrm{e} = 2\mathrm{e}_\mathrm{o} + \mathrm{e}_\mathrm{e}$$

then the bit-wise AND and XOR of a and b is given by the following

$$\mathsf{a} \wedge \mathsf{b} = 2\mathrm{o}_\mathrm{o} + \mathrm{e}_\mathrm{o} \qquad \mathsf{a} \oplus \mathsf{b} = 2\mathrm{o}_\mathrm{e} + \mathrm{e}_\mathrm{e}$$

it is then sufficient to check $\mathsf{c} = 2\mathrm{o}_\mathrm{o} + \mathrm{e}_\mathrm{o}$ for checking $\mathsf{a} \wedge \mathsf{b} = \mathsf{c}$.

5 Proofs for the Correct Program Execution over the ILC Channel

In this section we give an overview of our proof system for correct TinyRAM program execution over the ILC channel by giving a breakdown of it into simpler proofs, which are detailed in the full paper [BCG+18]. Recall that in the idealised linear commitment channel ILC the prover can submit `commit` commands to commit vectors of field elements of length k. The vectors remain secretly stored in the channel. The verifier can do two things: it can use a `send` command to send a message to the prover; and it can submit `open` queries to the ILC channel for obtaining the openings of linear combinations of vectors committed

by the prover. The field $\mathbb{F}$ and the vector length k are specified by the public parameter pp_{ILC}. It will later emerge that the best communication and computation complexity for a TinyRAM program terminating in T is achieved when k is approximately $\sqrt{T}$.

In Sect. 3 we broke the relation of correct program execution down to a number of sub-relations defined over a finite field $\mathbb{F}$. Our strategy for proving that they are all satisfied is to commit the extended witness to the ILC channel and then give an sub-proofs for each sub-relation. To begin we describe how we commit to the execution trace to the ILC model and discuss a relation $\mathcal{R}_{\mathsf{format}}$ for checking that the commitments are well formed. We then take a top down approach in order to describe how to check in the ILC model that the program has been executed correctly. In the first layer we describe a proof for correct TinyRAM execution in the ILC model. This proof decomposes into proofs checking that $\mathcal{R}_{\mathsf{check}}$, $\mathcal{R}_{\mathsf{mem}}$, $\mathcal{R}_{\mathsf{step}}$, and $\mathcal{R}_{\mathsf{format}}$ all hold. In the second layer we then decompose proofs for $\mathcal{R}_{\mathsf{format}}$, $\mathcal{R}_{\mathsf{check}}$, $\mathcal{R}_{\mathsf{mem}}$, and $\mathcal{R}_{\mathsf{step}}$ in terms of generic proofs for checking relations $\mathcal{R}_{\mathsf{const}}$, $\mathcal{R}_{\mathsf{perm}}$, $\mathcal{R}_{\mathsf{range}}$, $\mathcal{R}_{\mathsf{eq}}$, $\mathcal{R}_{\mathsf{blookup}}$ and $\mathcal{R}_{\mathsf{lookup}}$. In the third layer we detail how these proofs decompose into proofs in ILC for elemental relations, such as sums, products, shifts, entry-products and grand-sums of committed vectors. Our fourth and final layer will provide proofs in the ILC for these elemental relations.

5.1 Commitments to the Tables

In our proof system, the prover first commits to the extended witness w. The extended witness includes the field elements in the execution table Exe, the memory table Mem, the program table Prog, the range table $\mathsf{EvenBits}$ and the exponent table Pow. The prover arranges these tables in multiple matrices and to their rows.

The prover commits to each column of the execution table (such as the T entries containing the time t, the T entries containing the programt counter pc_t, etc.) by arranging it into an ℓ by k matrix, and making a commitment to each row of the resulting matrix. Entries of Exe relative to the same TinyRAM state will be inserted in the same position across the different matrices. Furthermore, in all these matrices the last entry of each column is duplicated in the first entry of the next column. As an example, let consider the first column of Exe which contains field elements representing the time-steps of the execution. Without loss of generality let $T = (\ell - 1)k + 1$, where T is the number of steps executed by the program and k is the vector length of the ILC. The prover organizes the field elements representing time in a matrix $\mathsf{E}_t \in \mathbb{F}^{\ell \times k}$

$$\mathsf{E}_t = \begin{pmatrix} 1 & \ell & 2\ell - 1 \dots & \\ 2 & \ell + 1 & 2\ell & \dots \\ \vdots & & & \ddots \\ \ell - 1 & 2\ell - 2 & 3\ell - 3 \dots & (\ell - 1)k \\ \ell & 2\ell - 1 & 3\ell - 2 \dots & T \end{pmatrix}$$

Similarly, the prover organizes the rest of the Exe table into matrices $\mathsf{E}_{\mathsf{pc}}, \mathsf{E}_{\mathsf{inst}}, \mathsf{E}_A, \ldots$ one for each column. Let E be the matrix obtained by stacking all matrices on top of each other and let $\mathsf{E} = \{\boldsymbol{e}_i\}$, for $\boldsymbol{e}_i \in \mathbb{F}^k$. The prover commits to Exe by sending the command $(\texttt{commit}, \{\boldsymbol{e}_i\}_i)$ to the ILC.

Each column of the program table is also committed to the ILC separately. In case $L \leq k$ we can store each column of Prog in one vector, i.e.

$$\mathsf{P} = \begin{pmatrix} \mathsf{P}_{\mathsf{pc}} \\ \mathsf{P}_{\mathsf{inst}} \\ \mathsf{P}_A \\ \cdots \end{pmatrix} = \begin{pmatrix} 0 & 1 & \ldots & L-1 \\ \mathsf{inst}_0 & \mathsf{inst}_1 & \ldots & \mathsf{inst}_{L-1} \\ A_0 & A_1 & \ldots & A_{L-1} \\ \cdots & & \cdots & \end{pmatrix}$$

otherwise, multiple rows can be used. The prover sends $(\texttt{commit}, \{\mathsf{P}_{\mathsf{pc}}, \mathsf{P}_{\mathsf{inst}}, \ldots\})$ to the ILC channel to commit to P.

The memory table Mem, the auxiliary lookup table EvenBits and the exponent table Pow can be committed in a similar way using matrices M, R and S

$$\mathsf{M} = \begin{pmatrix} \mathsf{M}_0 \\ \mathsf{M}_1 \end{pmatrix} \quad \mathsf{R} = \begin{pmatrix} 0\ 1\ 4\ 5 \ldots \sum_{i=0}^{\frac{W}{2}-1} k_i 2^{2i} \\ \ddots \\ \sum_{i=0}^{\frac{W}{2}-1} 2^{2i} \end{pmatrix} \quad \mathsf{S} = \begin{pmatrix} 0\ 1\ 2\ 3 \ldots W-1\ W \\ 1\ 2\ 4\ 8\ \ddots\ 2^{W-1}\ \ 0 \end{pmatrix}$$

where

$$\mathsf{M}_0 = \begin{pmatrix} \mathsf{M}_{\mathsf{addr},0} \\ \mathsf{M}_{\mathsf{v},0} \\ \mathsf{M}_{\mathsf{usd},0} \end{pmatrix} = \begin{pmatrix} 0 & 1 & \ldots & M-1 \\ \mathsf{v}_0 & \mathsf{v}_1 & \ldots & \mathsf{v}_{M-1} \\ 0 & 0 & \ldots & 0 \end{pmatrix} \quad \mathsf{M}_1 = \begin{pmatrix} \mathsf{M}_{\mathsf{addr},1} \\ \mathsf{M}_{\mathsf{v},1} \\ \mathsf{M}_{\mathsf{usd},1} \end{pmatrix} = \begin{pmatrix} 0 & 1 & \ldots & M-1 \\ \mathsf{v}_0 & \mathsf{v}_1 & \ldots & \mathsf{v}_{M-1} \\ 1 & 1 & \ldots & 1 \end{pmatrix}$$

and $(k_{\frac{W}{2}-1}, \ldots, k_0)$ is the binary expansion of k.

In order to show that the tables are committed to in the above manner the prover will show that the first row each of the matrices describing [Exe] is a shift the last row.

$$\mathcal{R}_{\mathsf{format}} = \left\{ \begin{array}{l} (pp, u, w) = ((W, K, \mathbb{F}, *), [\mathsf{E}], \perp) \ \Big| \\ \text{for } 1 \leq j \leq k-1 : [\mathsf{E}]_{\ell,j} = [\mathsf{E}]_{1,j+1} \end{array} \right\}$$

5.2 Proof for Correct TinyRAM Execution in the ILC Model

Given the witness for the correct execution of a TinyRAM program, we now describe how a prover can use the ILC channel to convince a verifier that the trace satisfies the relation $\mathcal{R}^{\mathsf{field}}_{\mathsf{TinyRAM}}$ corresponding to the correct program execution. The prover and verifier are given in Fig. 4.

Theorem 1. *$(\mathcal{K}_{\mathsf{ILC}}, \mathcal{P}_{TinyRAM}, \mathcal{V}_{TinyRAM})$ is a proof system for $\mathcal{R}^{\mathsf{field}}_{\mathsf{TinyRAM}}$ over the ILC channel with perfect completeness, statistical knowledge soundness with straight-line extraction, and perfect special honest-verifier zero-knowledge.*

$\mathcal{P}_{\text{TinyRAM}}(pp_{\mathsf{ILC}}, u, w)$	$\mathcal{V}_{\text{TinyRAM}}(pp_{\mathsf{ILC}}, u)$
– Parse $u = (P, v, T, M)$. – Extend w to w and parse it as $\left\{\begin{matrix}\mathsf{E}_t, \mathsf{E}_{\mathsf{pc}}, \mathsf{E}_{\mathsf{inst}}, \mathsf{E}_A, \mathsf{E}_{\mathsf{reg}_0}, \ldots, \mathsf{E}_{\mathsf{reg}_{K-1}}, \ldots \\ \mathsf{P}_{\mathsf{pc}}, \mathsf{P}_{\mathsf{inst}}, \ldots, \mathsf{M}_{0,\mathsf{addr}}, \mathsf{M}_{0,\mathsf{val}}, \ldots, \mathsf{R}\end{matrix}\right\}$ as in Section 5.1. Commit w in this form to the ILC channel. – $\mathcal{P}_{\text{check}}(pp_{\mathsf{ILC}}, u, \mathsf{w})$ – $\mathcal{P}_{\text{mem}}(pp_{\mathsf{ILC}}, u, \mathsf{w})$ – $\mathcal{P}_{\text{step}}(pp_{\mathsf{ILC}}, u, \mathsf{w})$ – $\mathcal{P}_{\text{format}}(pp_{\mathsf{ILC}}, u, \mathsf{w})$	– parse $u = \{[\mathsf{E}], [\mathsf{P}], [\mathsf{M}], [\mathsf{R}]\}$ – $\mathcal{V}_{\text{check}}(pp_{\mathsf{ILC}}, u)$ – $\mathcal{V}_{\text{mem}}(pp_{\mathsf{ILC}}, u)$ – $\mathcal{V}_{\text{step}}(pp_{\mathsf{ILC}}, u)$ – $\mathcal{V}_{\text{format}}(pp_{\mathsf{ILC}}, [\mathsf{E}])$ – Return 1 if all checks pass Return 0 otherwise

Fig. 4. Proof of correct TinyRAM execution in the ILC model

Proof. Perfect completeness follows from the perfect completeness of the sub-proofs. Perfect SHVZK follows from the perfect SHVZK of the sub-proofs. A simulated transcript is obtained by combining the outputs of the simulators of all the sub-proofs. Statistical knowledge soundness follows from the knowledge soundness of the sub-proofs. Since all sub-proofs have knowledge soundness with straight-line extraction, so does the main proof. □

The efficiency of our TinyRAM proof in the ILC model is given in Fig. 5. The asymptotic results displayed below are obtained when the parameter k specified by pp_{ILC} is approximately $\sqrt{T}$. The query complexity qc is the number of linear combinations the verifier queries from the ILC channel in the opening query. The verifier communication C_{ILC} is the number of messages sent from the verifier to the prover via the ILC channel and in our proof system it is proportional to the number of rounds. Let μ be the number of rounds in the ILC proof and $t_1, \ldots, t_\mu$ be the numbers of vectors that the prover sends to the ILC channel in each round, and let $t = \sum_{i=1}^{\mu} t_i$.

Prover computation	$T_{\mathcal{P}_{\mathsf{ILC}}} = \mathcal{O}(T)$ multiplications in $\mathbb{F}$
Verifier computation	$T_{\mathcal{V}_{\mathsf{ILC}}} = \text{poly}(\lambda)(L + \lvert v\rvert + \sqrt{T})$ multiplications in $\mathbb{F}$
Query complexity	$\text{qc} = \mathcal{O}(1)$
Verifier communication	$C_{\mathsf{ILC}} = \mathcal{O}(\log\log T)$ field elements
Round complexity	$\mu = \mathcal{O}(\log\log T)$
Total number of committed vectors	$t = \mathcal{O}\left(\sqrt{T}\right)$ vectors in $\mathbb{F}^k$

Fig. 5. Efficiency of our TinyRAM proof in the ILC model for $(pp, u, w) \in \mathcal{R}_{\mathsf{TinyRAM}}$. Here we are assuming that the number of instructions and words of memory $L, M < \sqrt{T}$, and that the number of registers K is constant.

6 Proofs for the Correct Program Execution over the Standard Channel

In the previous section we gave an efficient SHVZK proof of knowledge over the ILC channel for correct TinyRAM program execution. We now want to give a SHVZK proof of knowledge for correct TinyRAM program execution in the standard communication model where messages are exchanged directly between prover and verifier. To do this, we use the compiler from Bootle et al. [BCG+17] who use an error-correcting code and a collision-resistant hash function to compile a zero-knowledge proof over the ILC channel to a zero-knowledge proof over the standard communication channel. We refer to the full paper [BCG+18] for a transformation to turn SHVZK proofs into ones achieving full-zero knowledge, and for a recursive approach for reducing the verification time of our proofs.

From ILC to the Standard Channel. The compiler from Bootle et al. [BCG+17] uses an hash function to instantiate a non-interactive commitment scheme which realizes the commitment functionality of the ILC in the standard model. The compilation relies on a common reference string that specifies an error-correcting code and the hash function. However, the common reference string is instance-independent and can be reused for several proofs. Moreover, it can be generated from uniformly random bits in $\mathrm{poly}(\lambda)(L+M+\sqrt{T})$ TinyRAM steps and has similar size, so it has little effect on the overall performance of the system. The following theorem follows directly from their work.

Theorem 2 (Bootle et al. [BCG+17]). *Using a linear-distance linear error-correcting code and a statistically-hiding commitment scheme, we can compile a public-coin straight-line extractable proof $(\mathcal{K}_{\mathsf{ILC}}, \mathcal{P}_{\mathsf{ILC}}, \mathcal{V}_{\mathsf{ILC}})$ for a relation $\mathcal{R}$ over the ILC channel to a proof $(\mathcal{K}, \mathcal{P}, \mathcal{V})$ for $\mathcal{R}$ over the standard channel. The compilation is computationally knowledge sound, statistically SHVZK, and preserves perfect completeness of the ILC proof.*

Combining the above with Theorem 1 we get our main theorem.

Theorem 3 (Main Theorem). *Compiling the ILC proof system $(\mathcal{K}_{\mathsf{ILC}}, \mathcal{P}_{TinyRAM}, \mathcal{V}_{TinyRAM})$ of Fig. 4, we get a proof system over the standard channel for the relation $\mathcal{R}^{\mathsf{field}}_{\mathsf{TinyRAM}}$ with perfect completeness, statistical SHVZK, and computational knowledge soundness assuming the existence of collision-resistant hash functions.*

In the following section we detail the efficiency of the proof system obtained by compiling the proof system of Fig. 4.

Efficiency of the compiled TinyRAM Proof System. Computation is feasible only when it is polynomial in the security parameter, i.e., $T = \mathrm{poly}(\lambda)$ and $M = \mathrm{poly}(\lambda)$. Assuming $T, M \geq \lambda$, this means $\log T = \Theta(\log \lambda)$ and $\log M = \Theta(\log \lambda)$. To address all memory we therefore need $W = \Omega(\log \lambda)$. To keep the circuit size of a processor modest, it is reasonable to keep the word size low, so we will assume $W = \Theta(\log \lambda)$. Our proof system also works for larger

word size but it is less efficient when the word size is superlogarithmic. Note that we can at the cost of a constant factor overhead store register values in memory and therefore without loss of generality assume $K = \mathcal{O}(1)$.

To get negligible knowledge error we need the field to have superpolynomial size $|\mathbb{F}| = \lambda^{\omega(1)}$. This means we need a superconstant ratio $e = \frac{\log |\mathbb{F}|}{W} = \omega(1)$. On a TinyRAM machine, field elements require e words to store and using school book arithmetic field operations can be implemented in $\alpha = \mathcal{O}(e^2)$ steps[7].

Our proof system is designed for a setting where the running time is large, so we will assume $T \gg L + M$. In the ILC proof for correct program execution the prover commits to $\mathcal{O}(T)$ field elements and uses $\mathcal{O}(T)$ field operations. The verifier on the other hand, only uses $\mathcal{O}(L + |v| + \sqrt{T})$ field operations.

To compile the ILC proof into a proof over the standard channel, Bootle et al. use a linear-time collision-resistant hash function and linear error-correcting codes. The collision-resistant hash function by Applebaum et al. [AHI+17] based on the bSVP assumption for sparse matrices is computable in linear time and can be used to instantiate the statistically hiding commitment scheme used in the compilation. As the hash function operates over bit-strings we need to ensure that the efficiency is preserved once implemented in a TinyRAM program. If we stored each bit in a separate word of size $W = \Theta(\log \lambda)$ we would incur a logarithmic overhead. However, the hash function is computable by a linear-size boolean circuit and we can therefore apply a bit-slicing technique. We view the hash of an n-word string as W parallel hashes of n-bit strings. Each of the bit-strings is processed with the same boolean circuit, which means they can computed in parallel in one go by a TinyRAM program using a linear number of steps.

The error-correcting codes by Druk and Ishai [DI14] have constant rate and can be computed with a linear number of field additions. Applying the error-correcting codes therefore only changes the prover and verifier complexities by constant factors during the compilation. This means the compilation preserves the efficiency of the ILC proof up to constant factors. Taking into account the overhead of doing field operations, we summarize the efficiency of our proof system in Fig. 6.

	Field operations	TinyRAM operations
Prover Computation	$\mathcal{O}(T)$ operations in $\mathbb{F}$	$\mathcal{O}(\alpha T)$ TinyRAM steps
Verifier Computation	$\text{poly}(\lambda)(L + \|v\| + \sqrt{T})$ ops in $\mathbb{F}$	$\text{poly}(\lambda)(L + \|v\| + \sqrt{T})$ steps
Communication	$\text{poly}(\lambda)\sqrt{T}$ field elements	$\text{poly}(\lambda)\sqrt{T}$ words
Round Complexity	$\mathcal{O}(\log \log T)$	$\mathcal{O}(\log \log T)$

Fig. 6. Efficiency of our proof system for $\mathcal{R}_{\mathsf{TinyRAM}}$ under the assumption $W = \Theta(\log \lambda)$, $K = \mathcal{O}(1)$, $L + M < T \approx 2^W$, $k \approx \sqrt{T}$, and $\log |\mathbb{F}| = \Theta(\sqrt{\alpha}) \log \lambda$ for an arbitrarily small $\alpha = \omega(1)$.

[7] More sophisticated techniques such as FFT may reduce the cost of field multiplications to $\mathcal{O}(e \log e)$ steps, but if e is only slightly superconstant it will take a long time before the asymptotics kick in.

References

[AHI+17] Applebaum, B., Haramaty, N., Ishai, Y., Kushilevitz, E., Vaikuntanathan, V.: Low-complexity cryptographic hash functions. Electron. Colloq. Comput. Complex. (ECCC) **24**, 8 (2017)

[AHIV17] Ames, S., Hazay, C., Ishai, Y., Venkitasubramaniam, M.: Ligero: lightweight sublinear arguments without a trusted setup. In: CCS 2017 (2017)

[AS98] Arora, S., Safra, S.: Probabilistic checking of proofs: a new characterization of NP. J. ACM **45**(1), 70–122 (1998)

[BBC+17] Ben-Sasson, E., et al.: Computational integrity with a public random string from quasi-linear PCPs. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10212, pp. 551–579. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56617-7_19

[BCC+16] Bootle, J., Cerulli, A., Chaidos, P., Groth, J., Petit, C.: Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 327–357. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_12

[BCCT12] Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In: Innovations in Theoretical Computer Science (ITCS), pp. 326–349 (2012)

[BCCT13] Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: Recursive composition and bootstrapping for SNARKS and proof-carrying data. In: STOC 2013, pp. 111–120 (2013)

[BCG+13] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 90–108. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_6

[BCG+17] Bootle, J., Cerulli, A., Ghadafi, E., Groth, J., Hajiabadi, M., Jakobsen, S.K.: Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10626, pp. 336–365. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70700-6_12

[BCG+18] Bootle, J., Cerulli, A., Groth, J., Jakobsen, S.K., Maller, M.: Nearly linear-time zero-knowledge proofs for correct program execution. IACR Cryptology ePrint Archive (2018)

[BCS16] Ben-Sasson, E., Chiesa, A., Spooner, N.: Interactive oracle proofs. In: Hirt, M., Smith, A. (eds.) TCC 2016. LNCS, vol. 9986, pp. 31–60. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53644-5_2

[BCTV14a] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 276–294. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_16

[BCTV14b] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von neumann architecture. In: USENIX Security Symposium, pp. 781–796 (2014)

[BFM88] Blum, M., Feldman, P., Micali, S.: Non-interactive zero-knowledge and its applications (extended abstract). In: ACM Symposium on Theory of Computing, STOC 1998, pp. 103–112 (1988)

[BFR+13] Braun, B., Feldman, A.J., Ren, Z., Setty, S.T.V., Blumberg, A.J., Walfish, M.: Verifying computations with state. In: ACM SOSP, pp. 341–357 (2013)

[BG12] Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 263–280. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29011-4_17

[BP12] Bitansky, N., Paneth, O.: Point obfuscation and 3-round zero-knowledge. In: Cramer, R. (ed.) TCC 2012. LNCS, vol. 7194, pp. 190–208. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28914-9_11

[BSBTHR18] Ben-Sasson, E., Ben-Tov, I., Horesh, Y., Riabzev, M.: Scalable, transparent, and post-quantum secure computational integrity (2018). https://eprint.iacr.org/2018/046.pdf

[BSCG+13] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: Tinyram architecture specification, v0. 991 (2013)

[CD] Cramer, R., Damgård, I.: Zero-knowledge proofs for finite field arithmetic, or: can zero-knowledge be for free? In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 424–441. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055745

[CDG+17] Chase, M., et al.: Post-quantum zero-knowledge and signatures from symmetric-key primitives. In: ACM Conference on Computer and Communications Security, CCS 2017, pp. 1825–1842 (2017)

[CMT12] Cormode, G., Mitzenmacher, M., Thaler, J.: Practical verified computation with streaming interactive proofs. In: Innovations in Theoretical Computer Science, ITCS 2012, pp. 90–112 (2012)

[CTV15] Chiesa, A., Tromer, E., Virza, M.: Cluster computing in zero knowledge. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 371–403. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_13

[DI14] Druk, E., Ishai, Y.: Linear-time encodable codes meeting the Gilbert-Varshamov bound and their cryptographic applications. In: Innovations in Theoretical Computer Science, ITCS 2014, pp. 169–182 (2014)

[FNO15] Frederiksen, T.K., Nielsen, J.B., Orlandi, C.: Privacy-free garbled circuits with applications to efficient zero-knowledge. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 191–219. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_7

[FS90] Feige, U., Shamir, A.: Witness indistinguishable and witness hiding protocols. In: ACM Symposium on Theory of Computing, STOC 1990, pp. 416–426 (1990)

[Gen09] Gentry, C.: Computing on encrypted data. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) CANS 2009. LNCS, vol. 5888, pp. 477–477. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10433-6_32

[GGI+15] Gentry, C., Groth, J., Ishai, Y., Peikert, C., Sahai, A., Smith, A.D.: Using fully homomorphic hybrid encryption to minimize non-interative zero-knowledge proofs. J. Cryptol. **28**(4), 820–843 (2015)

[GGPR13] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 626–645. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_37

[GI08] Groth, J., Ishai, Y.: Sub-linear zero-knowledge argument for correctness of a shuffle. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 379–396. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78967-3_22

[GKR08] 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, Victoria, British Columbia, Canada, 17–20 May 2008, pp. 113–122 (2008)

[GMO16] Giacomelli, I., Madsen, J., Orlandi, C.: Zkboo: faster zero-knowledge for boolean circuits. In: 25th USENIX Security Symposium, pp. 1069–1083 (2016)

[GMR85] Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems (extended abstract). In: ACM Symposium on Theory of Computing, STOC 1985, pp. 291–304 (1985)

[Gro09] Groth, J.: Linear algebra with sub-linear zero-knowledge arguments. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 192–208. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03356-8_12

[Gro10a] Groth, J.: Short pairing-based non-interactive zero-knowledge arguments. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 321–340. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17373-8_19

[Gro10b] Groth, J.: A verifiable secret shuffle of homomorphic encryptions. J. Cryptol. **23**(4), 546–579 (2010)

[Gro16] Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 305–326. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_11

[IKO07] Ishai, Y., Kushilevitz, E., Ostrovsky, R.: Efficient arguments without short PCPs. In: IEEE Conference on Computational Complexity, CCC 2007, pp. 278–291 (2007)

[IKOS09] Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge proofs from secure multiparty computation. SIAM J. Comput. **39**(3), 1121–1152 (2009)

[JKO13] Jawurek, M., Kerschbaum, F., Orlandi, C.: Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In: ACM Conference on Computer and Communications Security, CCS 2013, pp. 955–966 (2013)

[Kil92] Kilian, J.: A note on efficient zero-knowledge proofs and arguments (extended abstract). In: ACM Symposium on Theory of Computing, STOC 1992, pp. 723–732 (1992)

[KR08] Kalai, Y.T., Raz, R.: Interactive PCP. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5126, pp. 536–547. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70583-3_44

[Nef01] Neff, C.A.: A verifiable secret shuffle and its application to e-voting. In: ACM Conference on Computer and Communications Security, CCS 2001, pp. 116–125 (2001)

[PHGR16] Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. Commun. ACM **59**(2), 103–112 (2016)

[Sch91] Schnorr, C.-P.: Efficient signature generation by smart cards. J. Cryptol. **4**(3), 161–174 (1991)

[Tha13] Thaler, J.: Time-optimal interactive proofs for circuit evaluation. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 71–89. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_5

[WHG+16] Wahby, R.S., Howald, M., Garg, S., Shelat, A., Walfish, M.: Verifiable ASICs. In: IEEE Symposium on Security and Privacy, SP 2016, pp. 759–778 (2016)

[WJB+17] Wahby, R.S., et al.: Full accounting for verifiable outsourcing. In: ACM Conference on Computer and Communications Security, CCS 2017, pp. 2071–2086 (2017)

[WSR+15] Wahby, R.S., Setty, S.T.V., Ren, Z, Blumberg, A.J., Walfish, M.: Efficient RAM and control flow in verifiable outsourced computation. In: Network and Distributed System Security Symposium, NDSS 2015 (2015)

[WTas+17] Wahby, R.S., Tzialla, I., Shelat, A., Thaler, J., Walfish, M.: Doubly-efficient zkSNARKs without trusted setup. Cryptology ePrint Archive, Report 2017/1132 (2017). https://eprint.iacr.org/

[ZGK+17] Zhang, Y., Genkin, D., Katz, J., Papadopoulos, D., Papamanthou, C.: vSQL: verifying arbitrary SQL queries over dynamic outsourced databases. In: IEEE Symposium on Security and Privacy, SP 2017, pp. 863–880 (2017)

[ZGK+18] Zhang, Y., Genkin, D., Katz, J., Papadopoulos, D., Papamanthou, C.: vRAM: faster verifiable ram with program-independent preprocessing (2018)