MiniRAM: VOLE-based ZKPoK for program executions

Martin Zacho, 201507667

Master's Thesis, Computer Science

June 2024

Advisor: Peter Scholl

AARHUS UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Abstract

Given a program p and time bound t, I consider the problem of proving in zero-knowledge the existence of an input x, such that running p on x returns 0 within t steps of execution. I give the syntax and operational semantics of a 32-bit RISC machine with a Harvard-architecture, named MiniRAM, and define the class of programs MRSAT_t that are *t-satifiable*.

I demonstrate that it's practical to prove, in zero-knowledge, t-satifiability of programs with a few hundred lines of code for $t \approx 30.000$ steps, taking 71 seconds for $\mathcal{P}$ and 19 seconds for $\mathcal{V}$ on a i7-9850H CPU with 16GB RAM, excluding pre-processing which makes use of a trusted random VOLE dealer. To achieve this, I construct a circuit compiler from MiniRAM to extended arithmetic circuits over $\mathbb{Z}_{2^{32}}$, using the CPU-verification approach originating in TinyRAM [BSCG$^+$13], and a VOLE-based zero-knowledge proof of knowledge (ZKPoK) adapted from QuarkSilver [BBMHS22], to prove satisfiability of circuits over $\mathbb{Z}_{2^k}$.

To evaluate if MiniRAM is practical for expressing cryptographic algorithms, that are otherwise tedious to express as circuits, I implement SHA256 in MiniRAM, and use my circuit compiler and ZKPoK to prove knowledge of a SHA256 pre-image for a given message digest in zero-knowledge. For a single round of hashing this takes 48 seconds for $\mathcal{P}$ and 8.6 seconds for $\mathcal{V}$.

Martin Zacho
Aarhus, June 2024.

Notation

- $\mathbb{P}(X)$ denotes the power-set of X, i.e. the set of all subsets of X.
- $\vec{x}$ denotes a vector. I will use $\vec{x}[i]$ or $\vec{x}_i$ to denote the i'th entry of $\vec{x}$.
- $\mathbb{Z}_n$ for $n \in \mathbb{N}^+$ denotes the set $\{0,1,2,...,n-1\}$ and $\mathbb{Z}_n^* = \{x \in \mathbb{Z}_n \mid gcd(x,n) = 1\}$, where gcd is the greatest common divisor of x and n.
- $a \cdot b$ can mean multiplication in a ring R if $a, b \in R$. If $a \in R$ and $\vec{b} \in R^t$ then $a \cdot b$ means scalar multiplication.
- For a natural number i then $[i]$ denotes the set $\{0,1,2,...,i-1\}$. I will also use $[i]$ to mean a secret sharing of i, but it will be clear from the context what meaning $[\cdot]$ has.
- I use $[0;1]$ to mean the subset of the real numbers from 0 to 1, i.e. $\{x | 0 \leq x \leq 1, x \in \mathbb{R}\}$.
- I use $\log x$ to mean the base-2 logarithm of x.
- I use $x \gg i$ to mean bitwise shifting x right by i.
- For any real number x, then $\lceil x \rceil \in \mathbb{Z}$ is the least integer greater than x.
- For strings $x, y \in \{0,1\}^*$ I will use $x || y \in \{0,1\}^*$ to mean the concatenation of x and y. Sometimes I will apply this operator to natural numbers, in which case I mean the concatenation of the binary encoding of the numbers.

Contents

Chapter 1

Introduction

Zero-knowledge proofs enable one party, the prover, to demonstrate towards another party, the verifier, that some statement is true without revealing any more information beyond this fact. The concept of zero-knowledge was first introduced by Goldwasser, Micali and Rackoff [GMR85], and in chapter 2 I will summarize their formalization of zero-knowledge proof systems, as well as variations such as interactive arguments and proofs of knowledge, to give the reader the necessary background knowledge that is foundational for the present thesis.

An important application of zero-knowledge proof systems in theory, as well as practice, is as sub-protocols in secure multiparty computation (MPC), for instance in voting schemes, or by making any passively secure protocol secure against active attacks [Gol04]. A long line of work has shown that it's feasible to design practical zero-knowledge arguments of knowledge for circuit satifiability (*CSAT*), for proving knowledge of a witness for a given *CSAT* instance in zero-knowledge, by using techniques such as garbled circuits [JKO13] [HK20b], MPC-in-the-head [AHIV23] or SNARKs [MBKM19]. However, my starting point has been the recent advances in zero-knowledge arguments of knowledge based on Vector Oblivious Linear Evaluations (VOLE), such as [WYKW20] [DIO20] [BMRS20] [YSWW21].

VOLE is a tool from secure two-party computation (2PC) and section 3.2 from chapter 3 gives the necessary background information needed to understand the VOLE-based zero-knowledge protocols. These protocols offer attractive features [BDSW23] such as

- **Fast prover** Authors of [BMRS20] report that a multi-threaded prover in Mac'n'Cheese is only 50 times at evaluating AES, than a single-threaded execution of an optimized AES implementation in C.
- **Linear memory overhead** The protocols only add a linear overhead on the memory requirements of the prover/ verifier, compared to evaluating the circuit in clear.
- **Post quantumn security** Security of the protocols relies on variants of the learning parity with noise (LPN) assumption, and thus believed to be resistant to quantumn attacks, as opposed to cryptographic protocols relying on factoring or discrete log.
- **Conceptual simplicity** Following a pre-processing phase, where the two parties agree on

correlated randomness (as explained in 3.2), then the protocols follow the established *commit-and-prove* [CD96] paradigm, which, for all practical purposes, is very simple to describe.

However, state-of-the-art VOLE-based zero-knowledge protocols all prove circuit satisfiability, and thus statements need to be formulated as circuits, usually with values from a large, finite field. This can be problematic in practice, since statements with an algorithmic character (as opposed to algebraic), such as statements about graph traversals or the output of hash functions, are tedious to express as arithmetic circuits, especially if the algorithm operates on machine words and the circuit is over a finite field.

The focus of this thesis is on proving satisfiability of programs written in a toy assembly language named MiniRAM. The syntax and semantics of the language is given in chapter 4, where I also formalize the notion of satisfiability of programs. The approach I've taken originated in TinyRAM **??**, and recent circuit compilers, such as Cheesecloth [CHP^{+}23], BubbleRAM [HK20a] and [**?**] all build upon this idea, which relies on modeling the processor and memory as circuits. The prover will then demonstrate that each step of execution was correct according to the step relation of the operational semantics. Section 4.2 explains how the circuit compiler works.

1.1 Prerequisites

Turing machines and their languages I assume the reader is familiar with the standard notion of a Turing machine, as a deterministic (unless otherwise stated) recognition device T, that on input a binary string x makes a number of computational steps (moving its tape head etc, see [Mar03] for a formal definition), and either halts in an accepting or rejecting state, or loops infinitely. We say that T accepts x if halts and accepts on input x (when running a Turing machine T on input x we sometimes use the notation $T(x)$, similar to ordinary function application), and the language of T is $L_T = \{x \mid T \text{ accepts } x\}$.

L is said to be a P-language if there exists a single-tape *deterministic* Turing machine T running in polynomial time with $L_T = L$. Similarly, L is said to be a NP-language if there exists a single-tape *non-deterministic* Turing machine T running in polynomial time with $L_T = L$.

A decision problem, such as deciding whether a graph contains a Hamiltonian path or not, can be converted to language-recognition problem, through suitable (and simple) encodings from the domain of the problem to finite strings of 0's and 1's (the language of the problem). I will sometimes say that a decision problem, a just problem, can be recognized, or accepted, by a Turing machine, when I mean that the Turing machine accepts the associated binary language of the problem. See [Kar10] for details.

We refer to the complexity class P as those problems whose languages are P-languages. Similarly, we refer to the complexity class NP as those problems whose languages are NP-languages.

We say a Turing machine T is exponential- or polynomial-time, if T halts on every input and the number of execution steps of the machine is bounded by some exponential or polynomial function of the length of the input to the machine.

Probabilistic Turing machines are ordinary Turing machines equipped with an additional read-only tape that's filled with random bits at the start of each execution. When reading a random bit we sometimes say the machine flips a random coin.

Interactive Turing machines (ITMs) are probabilistic Turing machines that are equipped with two additional tapes for reading from/ writing to other interactive Turing machines.

We sometimes allow Turing machines access to other Turing machines in a "black-box" manner, as well as the ability to "rewind" other interactive Turing machines as part of their computation. One can conceptually think of this as the same thing as rebooting a laptop, putting it back into a more desirable state than it was before, in order to see what happens when inputting x instead of y to the machine.

Groups, rings and fields I assume the reader is familiar with the usual definitions of a group, ring and field (see [Lau03]). I will use $G = \langle g \rangle$ to mean the group generated by g and $|G|$ to mean its order.

I will use S_n to refer to the symmetric group of n elements with finite order $n!$. For a permutation $\sigma \in S_n$ and vector $\vec{x} \in U^n$, with entries from some set U, I will abuse notation slightly and use $\sigma(\vec{x}) \in U^n$ to mean the vector resulting from applying σ to each entry in $\vec{x}$, i.e $\sigma(\vec{x})_i = \vec{x}_{\sigma(i)}$.

I will also assume that the reader is familiar with the discrete log (DL) and Diffie-Hellmann (DH) problems. This is not necessary to understand the main sections of the thesis, but I sometimes use them in examples in section **??**, and repeat them here for convenience: The DL problem is this: Given a group $G = \langle g \rangle$ and $g^n \in G$, compute n. The DH problem is slightly different: Given a group $G = \langle g \rangle$ and $g^x, g^y \in G$ compute $g^{xy} \in G$.

(In)distinguishability of probabilistic functions I assume the reader is familiar with the concept of negligible probabilities and indistinguishability of probabilistic functions. If not, then here's a lightening quick introduction: A negligible function $f : \mathbb{N} \to \mathbb{R}$ from the natural to the real numbers, is a function whose values are "too small to matter" after the input passes a certain threshold. More concretely, f is said to be negligible (as a function of its input) if it is asymptotically bounded from above by a function of the form n^{-c} for some constant c, i.e for any positive polynomial p then there is a k, such that for all $n > k$ then $f(n) \leq 1/p(n)$.

For a probabilistic algorithm U let $U_x : \{0,1\}^* \to [0;1]$ be the probability distribution of the algorithms output, when run on $x \in \{0,1\}^*$. Two probabilistic algorithms U and V are said to be perfectly indistinguishable if $U_x = V_x$ for all x.

Passive and active security I assume the reader has some familiarity with the usual definitions of passive and active security definitions for 2PC in the stand-alone model (see **??**). Intuitively, a protocol is secure against a passive adversary, if whatever the adversary sees during execution of the protocol can be computed from the adversary's inputs and outputs. This is formalized by the simulation paradigm, which I also elaborate on in section 2.2. Active security is formalized using the ideal- and real-world executions, where the ideal world models the following idealized execution of the protocol: Each party sends their input to a trusted third-party (called the ideal functionality), that computes the function for them, and sends each party their result. Loosely speaking, a protocol is secure against active attacks if the protocol emulates executions in the ideal world.

Commitment schemes A commitment scheme, which are cryptographic primitives used throughout secure MPC, consists of 3 algorithms: $Gen(\lambda) \to pp$ which generates a set of public parameters from a security parameter $\lambda \in \mathbb{N}$, $\mathit{CheckParams}(pp) \to \{0,1\}$ which checks if the parameters are well-formed, and $\mathit{Commit}_{pp}(x;r) \to c$ which generates a commitment to x with randomness r. The fundamental properties required of commitment schemes are *hiding* and *binding*, which informally means that the commitment shouldn't reveal anything about the message, and once a commitment has been created, the value, to which it is a commitment, cannot be changed.

A commitment scheme is said to be additively homomorphic if

$$\mathit{Commit}_{pp}(x;r_1) + \mathit{Commit}_{pp}(y;r_2) = \mathit{Commit}_{pp}(x+y;r_1+r_2)$$

whenever $\mathit{CheckParams}(pp) = 1$.

Oblivious Transfer *1-out-of-n* oblivious transfer (OT) is a 2-party protocol, where P_1 inputs $i \in [n]$, P_2 inputs $\vec{m} \in S^n$, for some domain S of messages, and by the end of the protocol P_1 learns $\vec{m}_i$ and P_2 learns nothing. Passively secure OT protocols can be constructed from public-key encryption (PKE) systems where public keys can be sampled pseudorandomly[1], and Chou and Orlandi have shown how to efficiently construct an actively secure 1-out-of-n OT protocol, assuming there exists groups where the computational Diffie-Hellmann problem (given a group G and $\beta^n \in G$ compute n) is hard [CO15], by tweaking the Diffie-Hellmann key exchange.

[1]in fact it's only necessary that there exists an alternative way of generating public keys that's indistinguishable from rreal keys and pre-image sampleable, see [OSB23].

Chapter 2

Zero-knowledge proof systems

Zero-knowledge proofs, introduced by Goldwasser, Micali and Rackoff in 1985 [GMR85], are powerful cryptographic primitives with broad applications across all of cryptography. They are used extensively in the field of secure multiparty computation (MPC) to make passively secure protocols secure against actively corrupted adversaries, which is an essential property for MPC protocols to have in a wide variety of settings.

In this chapter I will summarize the theoretical foundations of zero-knowledge proof systems. This includes describing the most broadly accepted formalizations of the notion of a mechanized theorem proving procedure, and how different variations of interactivity and computing resources manifest in these.

2.1 Interactive proof systems

The notion of a proof, like the notion of computation, is an intuitive one [GMR85]. Similar to how Turing machines have proven to be a good formalization of computations, then the theory of *NP* have proven to be a good formalization of the notion of a mechanized proof system.

Our starting point is therefore the influential definition of *NP* by Cook in 1971 [Coo71], here given in the formulation by Goldwasser, Micali and Rackoff:

Definition: **Proof system for *NP* language L**

A proof system for an *NP* language L consists of two ordinary Turing machines $\mathcal{P}$ and $\mathcal{V}$, named the prover and verifier respectively. The prover is exponential-time, the verifier is polynomial-time, and they both read a common input and interact in elementary ways.

On input string $x \in L$, $\mathcal{P}$ computes a string y (whose length is bounded by a polynomial in the length of x) and writes y on a special tape that $\mathcal{V}$ can read. $\mathcal{V}$ then checks that $f_L(y) = x$, where f_L is a polynomial-time computable function depending on the language L, and, if so, halts and accepts.

Intuitively, the function f_L verifies that $\mathcal{P}$ has output a valid witness for problem x. For example, given a group $G = \langle g \rangle$, consider the language L consisting of yes-instances of the Decisional

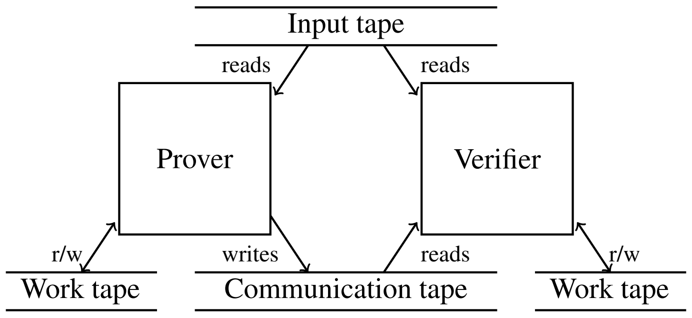


Figure 2.1: A proof system consisting of two Turing machines connected by a one-way communication tape, as defined in the monumental paper by Goldwasser, Micali and Rackoff [GMR85]

Diffie-Hellman problem, i.e encodings of three group elements α, β, γ for which $\alpha = g^a, \beta = g^b$ and $\gamma = g^{ab}$ for integers $a, b \in \mathbb{Z}_{|G|}$. On input an encoding of three group elements $\alpha = g^a, \beta = g^b$ and $\gamma = g^c$ the prover uses its exponential computational powers to compute the discrete logs $a, b, c \in \mathbb{Z}_{|G|}$, and then outputs $y = a||b||c$ on the special tape used to communicate with $\mathcal{V}$, who then runs $f_L(a, b, c)$ which checks that $c = ab, \alpha = g^a, \beta = g^b$ and $\gamma = g^c$.

For a proof system to model traditional mathematical proofs, it is natural to require that the prover should be able to "convince" the verifier of true statements, and that any (possibly malicious, or just plain ignorant) prover cannot "convince" the verifier of false statements. The verifier is "convinced" when it enters an accepting state at the end of the conversation. We interpret "false" and "true statements" as $x \notin L$ and $x \in L$ respectively.

These are the usual requirements of soundness and completeness, whose names have been borrowed from formal logic:

- *Completeness*: We say that a proof system for *NP* language L is complete, if for any $x \in L$ the prover outputs a string y that will make the verifier halt and accept.

- *Soundness*: We say that a proof system for *NP* language L is sound, if for any $x \notin L$ and exponential-time prover P^*, the prover outputs a string y that will make the verifier halt and reject.

It's absolutely essential for a proof system to be sound and complete, if it is to be of any use. Otherwise, true statements cannot be proven or false statements can be proven, which certainly are bad properties of a theorem proving procedure. Furthermore, matching our intuition of traditional mathematical proofs, verification is computationally easy, as the verifier is only allowed to run in polynomial-time of the length of the statement.

Proof systems model traditional mathematical proofs that can be communicated by "writing them down in a book". An alternative idea is that of an *interactive* proof system, which was also introduced by Goldwasser, Micali and Rackoff in [GMR85], as a way of formalizing the intuitive idea of a proof, that can be "explained in class". In this setting the lecturer can take advantage of the possibility of interacting with the "recipients" of the proof, who may ask questions at crucial times during the proof, to support their understanding any difficult parts. This makes lectures job of communicating the proof a lot easier, since details may be added upon request rather than up front.

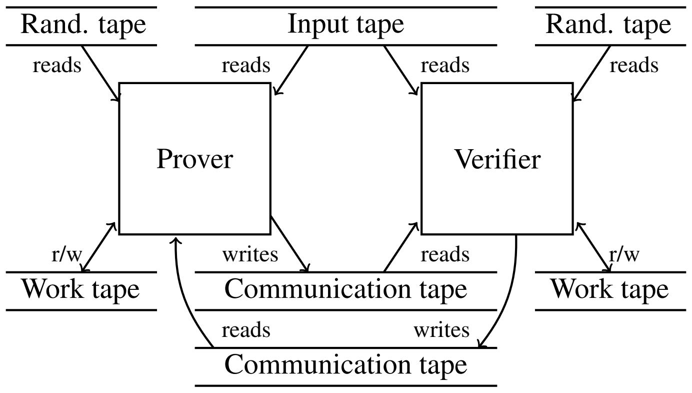


Figure 2.2: An interactive proof system consisting of two interactive (and thus probabilistic) Turing machines.

The formalization is similar to that of ordinary proof systems, but the machines are now configure with communication tapes:

Definition: **Interactive proof system**

Let $L \subseteq \{0,1\}^*$ be a binary language and $(\mathcal{P}, \mathcal{V})$ be two interactive (and thus probabilistic) Turing machines, configured to read from/ write to each others communication tapes. We say that they are an interactive proof system, if $\mathcal{P}$ has infinite power, $\mathcal{V}$ is polynomially bounded, and they satisfy the following properties:

- Completeness: For any $x \in L$ given as input to $(\mathcal{P}, \mathcal{V})$, $\mathcal{V}$ halts and accepts, except with negligible probability in the length of x.
- Soundness: For any $\mathcal{P}^*$, and any $x \notin L$ given as input to $(\mathcal{P}^*, \mathcal{V})$, $\mathcal{V}$ halts and rejects, except with negligible probability in the length of x.

The first condition again ensures that true statements are provable, and the second condition ensures that false statements cannot be proved, except with a probability that's overwhelmingly small.

2.2 Zero-knowledge and the Simulation Paradigm

Proving a mathematical theorem can be seen as a way of communicating, or transferring, knowledge from a prover to a receiver. The study of knowledge has a long and deep history, however, as this work is not a survey of different paradigms of epistemology, or the disagreements of the nature of knowledge, our definition of knowledge is the one widely agreed upon in the field of cryptographic computing. To the best of my knowledge, this point of view also originates from [GMR85]:

Knowledge is a notion relative to a specific model of computation with specified computing resources. One studies and gains knowledge about available objects.

In other words, if a computationally unbounded participant learns the product of two large primes $n = pq$, then that party learns p and q. On the other hand, assuming factoring is hard, then a participant that only has polynomially bounded computational resources does not learn p or q by learning n. So participants "know what they know" and what they can compute from what they know, and they gain knowledge by learning the output of computations they cannot perform themselves. Using the same principle, if you send me the result of ten fair coin flips, then this isn't new knowledge to me, since I could have sampled the uniform distribution on $\{0,1\}$ ten times myself.

This notion of knowledge is very handy when arguing that participants in a cryptographic protocol don't learn more than some logical statement X: If from X the participant can compute every piece of knowledge communicated to the participant during the protocol using, of course, only computational resources available to the participant, then the participant will have learned no more than X by engaging in the protocol. This is so, since the participant *already knew* what they see during the protocol, since they can compute it from X.

To formalize this in the context of interactive proof systems, we require that for each verifier that interacts with the prover, attempting to be convinced that some logical statement is true, there must exist a way for the verifier to compute a "transcript" all by herself, in a way that's indistinguishable from actual conversions with the prover, using only the polynomially bounded resources available to the verifier.

This formalization of an interacting theorem proving procedure, that reveals no more information than the validity of a logical statement, is again from [GMR85], here given in the formulation by Damgaard, Nielsen and Yakoubov [DNY23]:

Definition: **Zero-knowledge interactive proof system**

Let ($\mathcal{P}$, $\mathcal{V}$) be an interactive proof system for a *NP* language L. We say that the system is zero-knowledge, if, for every polynomial-time ITM $\mathcal{V}^*$, there is a probabilistic polynomial-time Turing machine M (called the *simulator*), such that for any input $x \in L$ to $(P, \mathcal{V}^*)$ and private input δ to $\mathcal{V}^*$, the communication between $\mathcal{P}$ and $\mathcal{V}^*$ (the final contents of their communication tapes) is indistinguishable from the output of $M(x)$.

The idea, as explained above, is that if the system is zero-knowledge, and a simulator M exists, then $\mathcal{V}^*$ can use M to generate a transcript that is distributed exactly as real conversations with $\mathcal{P}$. Thus $\mathcal{V}^*$ learns nothing else than $x \in L$.

The private input δ to the possibly dishonest verifier represents a priori information, that the verifier might have learned through previous communication with the prover.

2.3 Interactive arguments

An interactive argument, also sometimes called a computationally convincing argument [DP97], is like an interactive proof system, but where the prover is constrained to run in polynomial time. According to [DNY23], the simplest definition is this:

Definition: **Interactive argument for language L**

An interactive argument is an interactive proof system ($\mathcal{P}$, $\mathcal{V}$) where $\mathcal{P}$ receives a auxiliary input w and is constrained to run in probabilistic polynomial time. The soundness condition it must satisfy only concerns cheating provers that are polynomially bounded, and the completeness conditions is this:

- Completeness: For any $x \in L$ given as input to ($\mathcal{P}$, $\mathcal{V}$), there exists a private input w to $\mathcal{P}$, which is polynomially bounded by the length of x, such that $\mathcal{V}$ halts and accepts, except with negligible probability in the length of x.

As noted in [DNY23], the private input to $\mathcal{P}$ is necessary, since if the prover did not have some advantage over $\mathcal{V}$ then the verifier might just solve the problem on her own, instead of talking to the prover.

2.4 Proofs of knowledge

A variation of the previously described proof systems is one where the verifier is not only convinces of language membership, but also that the prover "knows" the associated *NP* witness. To formalize how one machine can convince that another machine "know" a piece of information, the idea is to require the existence of a way to compute the information from rewindable access to the convincing prover.

Proofs of knowledge were a conceptual contribution of [GMR85] and formally defined in [BG92], although we won't be using their definition in its exact form. It concerns the general setting of a computationally unrestricted prover, and we are only interested in practical provers, that can be implemented on todays hardware[1], and only for arguing knowledge of a witness for a particular *NP* relation.

We therefore take inspiration from [Dam23], which is also based on [BG92]:

Definition: **Zero-knowledge proof of knowledge for *NP* language L_R**

Let R be an *NP* relation and L_R its associated language, $\kappa : \{0,1\}^* \to [0;1]$, and ($\mathcal{P}$, $\mathcal{V}$) be polynomial-time interactive Turing machines, configured to read from/ write to each others communication tapes. We say that ($\mathcal{P}$, $\mathcal{V}$) is a zero-knowledge proof of knowledge (ZKPoK) for L with knowledge error κ if

- Completeness: For all $(x, w) \in R$, if x is given as input to ($\mathcal{P}$, $\mathcal{V}$) and w is given as private input to $\mathcal{P}$ then $\mathcal{V}$ halts and accepts, except with negligible probability.

- Knowledge soundness: There exists an polynomial-time interactive Turing machine M, called the *knowledge extractor*, which, on input x and rewindable black-box access to

[1]That is, we're concerned with *arguments* of knowledge that are zero-knowledge, but it seems most of the literature on practical systems call these proofs of knowledge, even though the prover has restricted resources, so I won't try to change that.

the prover attempts to compute a valid witness for x, that is, a w such that $(x,w) \in R$. We require the following: For any prover $\mathcal{P}^*$, let $\varepsilon(x)$ be the probability that $\mathcal{V}$ accepts on input x. There exists a constant c and such that whenever $\varepsilon(x) > \kappa(x)$ then M will output a valid witness for x in time bounded by

$$\frac{|x|^c}{\varepsilon(x) - \kappa(x)}$$

We sometimes refer to $\mathcal{V}$ as the knowledge verifier of the system, and κ as the knowledge error function.

The system formalizes the idea, that if any prover can convince the knowledge verifier that it "knows" a witness for some *NP* statement, then it must indeed "know" such a witness, except with a probability that's overwhelmingly small. The notion of knowledge is again computational knowledge, so if the prover manages to convince the verifier, then there must exist an efficient procedure for "pulling out the witness" from the prover, i.e computing a w for which $(x,w) \in R$ using black-box and rewindable access to $\mathcal{P}$. The running time of the knowledge extractor is proportional to the probability with which $\mathcal{P}$ convinces $\mathcal{V}$.

ZKPoK can be seen as a special case of general secure two-party computation, in which only one party (the prover) has a private input [JKO13]. Therefore, modern work on zero-knowledge happens within the context of MPC. For the purpose of formally defining the intended behavior and security guarantees of my protocol, I'll give the following ideal ZKPoK (from now on abbreviated ZK) functionality[2], which models the idealized behavior of the zero-knowledge protocol, for proving satisfiability of a kind of "extended" arithmetic circuits, which I'll give more details on in 3.1:

Functionality: **Zero-knowledge functionality $\boldsymbol{F^k_{\mathrm{ZK}}}$**

Prove On input $(\textsc{prove}, C, \vec{w})$ from $\mathcal{P}$ and $(\textsc{verify}, C)$ from $\mathcal{V}$, where C is an extended arithmetic circuit with n inputs over $\mathbb{Z}_{2^k}$ and $\vec{w} \in \mathbb{Z}^n_{2^k}$:

- If $\mathcal{P}$ is honest let $X = \vec{w}$. Otherwise ($\mathcal{P}$ is corrupt) receive $X \in \mathbb{Z}^n_{2^k}$ from $\mathcal{A}$.
- If $\mathcal{V}$ is honest set $\mathcal{R} = \mathcal{V}$. Otherwise ($\mathcal{V}$ is corrupt) set $\mathcal{R} = \mathcal{A}$.
- If evaluating C on X yields a collection of 0's send $(\textsc{true})$ to $\mathcal{R}$. Otherwise send $(\textsc{false})$ to $\mathcal{R}$.

Corrupt On input $(\textsc{corrupt}, X)$ from $\mathcal{A}$ with $X \in \{\mathcal{P}, \mathcal{V}\}$ corrupt X.

The functionality simply receives a witness $\vec{w}$ from the honest prover and sends $(\textsc{true})$ to the honest verifier only if $C(\vec{w}) = 0$, while allowing for the usual notion of *actively* corrupting a party by an adversary $\mathcal{A}$, taking complete control of all input/output between the functionality and the party.

[2] a functionality in the stand-alone model is a general algorithmic entity very similar to an interactive Turing machine, and in the setting of n-party secure MPC, then the ideal functionality constitutes the idealized computation of some function $f : \{0,1\}^n \to \{0,1\}^n$.

Chapter 3

VOLE and the QuickSilver protocol

As explained in the previous chapter, zero-knowledge proof systems are interactive machines, where the prover convinces the verifier that an instance of a language recognition problem is a yes-instance. Different systems can be used to prove membership in different languages, and for our purpose we are interested in systems where a valid witness can be extracted from the convincing prover.

In the first section of this chapter I will describe the kind of arithmetic circuits whose associated satisfiability problem *ECSAT* I am interested in proving membership in. The circuits have non-standard arithmetic gates that will be useful when building the circuit compiler for MiniRAM programs, for instance **Select** gates, which works similar to an array indexing operator, that are used for fetching instructions from memory and values from the machine's registers.

The basis for the ZK protocol I've implemented is that of QuickSilver [YSWW21], which is one of the cornerstones of recent advances in ZK proof systems due to its optimal memory footprint, and that it only needs to communicate 1 field element for each multiplication gate. I will describe the protocol in section 3.3.1.

Correlated randomness comes in different forms and is essential to many cryptographic protocols. As the simplest example of this, if two parties hold random strings x and y respectively, and the strings are correlated by being *equal*, then the parties are able to privately communicate any string with the same length as x and y, by using a one-time pad. As an example of a non-trivial correlation, random "multiplication triples" (also called beaver triples), where parties hold secret shares of a, b and ab, can be used to efficiently and securely compute xy for x and y held by the two parties respectively [Bea91]. In the second section of the chapter I will explain what Vector Oblivious Linear Evaluation (VOLE) is, and how it can be used to establish "linearly" correlated randomness, for securely computing an arithmetic circuit in the QuickSilver protocol.

An extension to QuickSilver is QuarkSilver [BBMHS22], which adapts QuickSilver to work over rings of the form $\mathbb{Z}_{2^k}$ instead of large prime fields. Since my extended arithmetic circuits are over $\mathbb{Z}_{2^{32}}$ my ZK protocol adapts QuarkSilver, rather than QuickSilver, to work over the non-standard gates of my circuits. Section 3.3.2 explains the problem with QuickSilvers multiplitaction checks when working in $\mathbb{Z}_{2^k}$, as well as how to fix by working in an extension ring, as it's been contributed in [BBMHS22] inspired by SPD$\mathbb{Z}_{2^k}$ [CDE$^+$18].

My protocol, which I've named QuarkSilver+, follows almost directly from QuarkSilver, however I've made a minor change to the sub-protocol for opening values in the circuit. The change is necessary to obtain the same probability as in QuarkSilver with which incorrect statements can be proven, since, in my protocol, more values than the last wire is opened[1], and they all contribute 2^{-s} to the soundness probability (by the union bound) in QuarkSilver. I give more details on this issue, and how I've solved it after presenting the protocol in figure 3.3.3.

The last section finally adapts the proof of security of QuarkSilver to QuarkSilver+. This is straight forward, but I'm given the security theorem in 3.3.4 and repeating the high-level idea of its proof. The full construction of the simulators is included in appendix A.

3.1 Extended arithmetic Circuits

For our purpose, an arithmetic circuit is a directed acyclic multigraph[2], where the nodes, also called *gates*, are labeled according to the operation they perform (such as addition, multiplication etc., but also non-standard gates, as I'll explain in a moment), and where the in-degree of a gate (the number of edges, also called *wires*, going into a gate) must adhere to the operation it performs. Multiplication gates, for instance, must have an in-degree of 2, so we will need 2 of these to construct a circuit that computes the function $f(x,y) = x^2y$. The first multiplication gate will have 2 incoming wires from the input node for x, which is why circuits are multigraphs rather than simple graphs. This is standard.

Most operations only compute a single value, such as addition or multiplication gates. However, some operations compute multiple values, such as the operation for deconstructing an integer into its bit-decomposition. To model this, the wires of the circuit are labeled with a natural number, the wire's *index*, signifying which of the values computed by the operation flows on that wire. All gates are allowed unbounded out-degree, so their outputs can be used multiple times by other gates, but the outgoing wires of a gate must have valid indices for the operation performed by the gate.

There's really only one thing you can do with a circuit, which is to evaluate it: Since circuits are acyclic they can be topologically sorted, and evaluating a circuit on $x_1,\ldots,x_n$ will, after assigning x_i to each output wire of input node i, consider each gate in topologically sorted order, and apply the gate's operation to its input wires to obtain a value for its output wires. There's nothing special going on here, this is all very standard. The output of a circuit is defined by the collection of all final values flowing into the output gates. We will not care about ordering the outputs in any meaningful way, as the prover is only interested in proving that they are all 0, for some assignment of input gates.

These are the kinds of operations computing a single value:

- **In**$_i$: "Computes" x_i, where x_i is supplied by the start of circuit evaluation.
- **Add**: On input $x_1,x_2,\ldots,x_n$, computes $\sum_{i=1}^{n} x_i$.
- **Sub**: On input x and y, computes $x-y$.

[1]For instance as part of the sub-protocols for **Select**, **Decode32** and **CheckAllEqButOne** gates.

[2]A multigraph is a graph that can have multiple (parallel) edges connecting two nodes. A directed multigraph is also sometimes called a multidigraph. In our case parallel edges all have distinct identity, as opposed to solely be being identified by the nodes they connect.

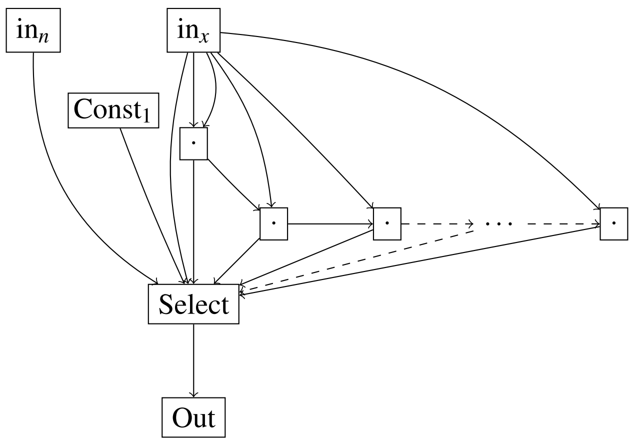


Figure 3.1: A simple arithmetic circuit computing the function $f(x,n) = x^n$ for $n \leq k$ where k is the number of multiplication gates in the circuit, demonstrating how Select gates add expressiveness to arithmetic circuits, that is convenient when constructing the MiniRAM-to-circuit compiler. All wires have implicit indices of 0 that aren't depicted, since they all just compute 1 output.

- **Mul**: On input x and y, computes xy.
- **Const**: "Computes" c, where c is a constant parameter of the gate.
- **MulConst**: On input x, the gate computes xc, where c is a constant parameter of the gate.
- **Select**: On input $i, x_0, x_1, ..., x_{n-1}$ the gate computes x_i. If $i \geq n$ then this produces an error and circuit evaluation stops[3].
- **SelectConst**: On input i the gate computes c_i, where $c_0, c_1, ..., c_{n-1}$ are constant parameters of the gate. If $i \geq n$ then this produces an error and circuit evaluation stops.
- **Encode4**: On input x_1, x_2, x_3, x_4 the gate computes $\sum_{i=1}^{4} 2^{i-1} x_i$. If any of the inputs cannot be represented by a bit, then this produces an error and circuit evaluation stops.
- **Encode8**: On input $x_1, x_2, ..., x_8$ the gate computes $\sum_{i=1}^{8} 2^{i-1} x_i$. If any of the inputs cannot be represented by a bit, then this produces an error and circuit evaluation stops.
- **Encode32**: On input $x_1, x_2, ..., x_{32}$ the gate computes $\sum_{i=1}^{32} 2^{i-1} x_i$. If any of the inputs cannot be represented by a bit, then this produces an error and circuit evaluation stops.

The only kind of operations computing multiple values, is the following one for decomposing an integer into its bit-decomposition:

- **Decode32**: On input x with $x < 2^{32}$ the gate computes x_1, ..., x_{32} such that $\sum_{i=1}^{32} 2^{i-1} x_i = x$.

These are the kinds of operations that don't compute any values, but has other sorts of side-effects during circuit evaluation:

[3]analogously to how out-of-bounds array accesses triggers a run-time error in a memory-safe language like Java.

- **Out**: On input x, the gate adds x to the collection of circuit outputs.
- **CheckAllEqButOne**: On input $i, x_0, y_0, x_1, y_1, ..., x_{n-1}, y_{n-1}$, the gate asserts that
  - $i < n$ and
  - $\forall j \in \{0, \ldots, n-1\} : j \neq i \implies x_j = y_j$.

  If that is not the case, then the gate produces an error and circuit evaluation stops[4].

Select gates perform an operation reminiscent of an array indexing in a programming language like Java, where out-of-bounds indexing results in a run-time error. It is rather powerful, as illustrated in figure 3.1, where it's used to easily compute the the power function for arbitrary exponents and bases, which is not obvious how to easily do with basic arithmetic circuits.

I've added a variant of the **Select** gates, namely **SelectConst**, which selects on constants. This addition is mainly for performance reasons, as any **SelectConst** can be replaced with a number of Const operations and a **Select**. Similarly, MulConst can obviously be replaced by Const and **Mul** operations.

CheckAllEqButOne gates perform a kind of assertion operation which doesn't produce any output, but will be very convenient for reasons that will be clearer when describing the part of the circuit compiler, that checks consistency of CPU registers at each computation step.

The encode and decode operations are each others inverses, and are useful for building circuits for instruction decoding, bitwise logical operations, division by 2 and such. As an example, a circuit for computing the logical AND of two 32-bit words using Decode32 and multiplication gates is shown in 3.2. I've also used them to build circuits for ripple-adders, in order to check if the addition of two 32-bit integers overflows, as well as compare two integers for equality and ordering.

For a circuit with any number of **Select** or **SelectConst** gates, I will use "the number of **Select** alternatives" to mean the sum of all the n's, for all **Select** or **SelectConst** gates with input $i, x_0, x_1, ..., x_{n-1}$.

Similarly, for a circuit with any number of **CheckAllEqButOne** gates, I will use "the number of **CheckAllEqButOne** pairs" to mean the sum of all the n's, for all **CheckAllEqButOne** gates with input $i, x_0, y_0, x_1, y_1, ..., x_{n-1}, y_{n-1}$.

Satisfiability of circuits The classic satisfiability problem for extended arithmetic circuits is this:

Definition: **Satisfiability of circuits *ECSAT***

Let C be a circuit over $\mathbb{Z}_{2^k}$ with n input gates. We call C *satisfiable* if the exist a vector $\vec{w} \in \mathbb{Z}_{2^k}^n$ such that evaluating C on $\vec{w}$ yields a collection of 0's. The decision problem *ECSAT* is to determine if a given circuit is satisfiable.

It is clear, that the functionality F_{ZK}^k is a system for proving that a circuit C is a yes-instance of *ECSAT*, and furthermore, that the prover knows the satisfying assignment (since it must be inputtet

[4]analogously to how assert statements work in most programming languages.

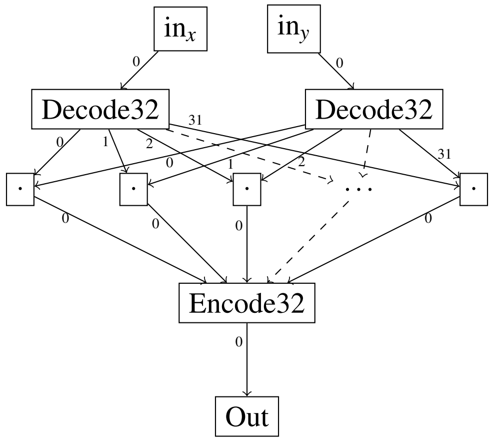


Figure 3.2: A circuit for computing the bitwise logical AND of two 32-bit words, using arithmetic operations. The labels on the outgoing edges of the Decode32 operations signify which output of the operation flows to the connected multiplication gate.

to the functionality). Also, since extended arithmetic circuits augments the usual arithmetic circuit, then the classic problem of arithmetic circuit satisfiability can be be reduced to *ECSAT*.

From now on, whenever I write “circuit”, I will mean an extended arithmetic circuit.

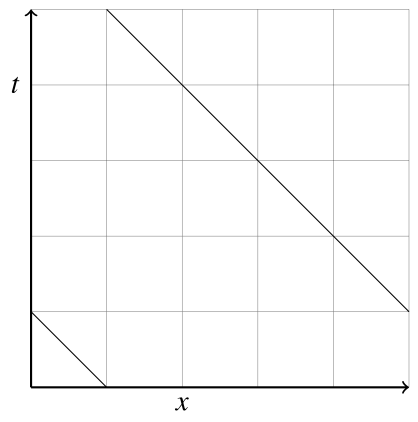
3.2 Vector Oblivious Linear Evaluation

Oblivious Linear Evaluation (OLE) is a secure two-party protocol where one party, the receiver, learns the evaluation of a secret linear function known to another party, called the sender. More precisely, for a finite field $\mathbb{F}$, a OLE *correlation* consists of a pair of random elements (t, x), $(\alpha, \beta) \in_R \mathbb{F}^2$ constrained to $t = \alpha x + \beta$. By interacting with an OLE protocol the receiver learns a random t for its choice of x, and the sender learns random α and β, subject to the linear constraint. Secure OLE protocols must ensure that the sender remains oblivious to the receiver's choice of x and thus evaluation t, while the receiver learns nothing about α and β held by the sender. The classic 1-out-of-2 Oblivous Transfer (OT), where a sender inputs messages m_0 and m_1 and a receiver inputs $i \in \{0, 1\}$ and learns m_i, can be seen as a specialization of OLE, by defining the linear function $f(x) = m_0 + x(m_1 - m_0)$ and constraining it to the domain $\{0, 1\}$.

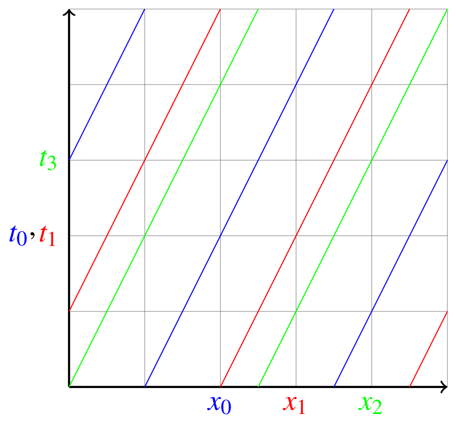
If two parties have access to a protocol securely implementing *random* OLE, where the receiver doesn't get to choose x, but learns a correlation for some random r, then the parties can easily establish a correlation for some x chosen by the receiver, at the cost of communication a single field element, with this simple protocol: After the receiver receives t and r it sends $\delta = x - r$ to the sender. The sender lets $\beta_x = \beta - \alpha\delta$, and now $t = \alpha x + \beta_x$ since

$$\alpha x + \beta - \alpha\delta = \alpha x + \beta - \alpha(x - r) = \alpha(x - x + r) + \beta = \alpha r + \beta = t$$

It is secure for the receiver to send δ, since r is random and unknown to the sender, thus from the point of view of the sender then $x - r$ is just a random field element (and a kind of one-time pad



(a) An Oblivious Linear Evaluation in $\mathbb{Z}_5$, where the sender learns $\alpha = 4$ and $\beta = 1$, and the receiver learns $x = 2$ and $t = 4$.



(b) A Vector OLE in $\mathbb{Z}_5$, where the sender learns $\Delta = 2$ and $\vec{\beta} = (3,1,0)$ and the receiver learns $\vec{x} = (2,3,4)$ and $\vec{t} = (2,2,3)$.

encryption of x, that can be decrypted by computing $\delta + r$).

An extension of OLE is Vector OLE (VOLE), where the receiver learns a linear combination of two random vectors held by the sender: Following an initial setup step, where the receiver learns a random $\Delta \in_R \mathbb{F}$, sometimes called the global key, the VOLE can be extended with n new *correlations*, where the receiver learns a random vector $\vec{t} \in_R \mathbb{F}^n$ for its choice of some vector $\vec{x}$, and the verifier learns the vector $\vec{\beta} \in \mathbb{F}^n$, constrained to $\vec{t} = \Delta\vec{x} + \vec{\beta}$.

Similarly to how two parties having access to a secure protocol for generating random OLE correlations can securely establish a OLE correlation for the senders choice of x, then, if two parties have access to a secure protocol for generating random VOLE correlations, they can, using the same protocol, easily and securely establish VOLE correlations for the senders choice of $\vec{x} \in \mathbb{F}^n$, albeit at the cost of communicating n field elements for the vector of deltas: After the receiver receives $\vec{t}$ and $\vec{r}$ it sends $\vec{\delta} = \vec{x} - \vec{r} \in \mathbb{F}^n$ to the sender, who computes $\vec{\beta}_x \in \mathbb{F}^n$ as $\vec{\beta} - \Delta\vec{\delta}$. The correctness and security arguments are exactly the same.

When extending the VOLE with a single correlation $t_i = \Delta x_i + \beta_i$, with x_i chosen by the receiver, this may be viewed as one party, in our case the prover, receiving a random tag t_i on its choice of x_i, produced by the key β_i only known to the sender, in our case the verifier. This is a kind of commitment scheme, where the prover commits to x, which is hidden from the verifier, and where commitments can be opened by the prover sending x_i, t_i to the verifier. The verifier then checks that $t_i = \Delta x_i + \beta_i$, and, if that's the case, accepts the opening. Therefore, we will sometimes also refer to a VOLE correlation as a random VOLE commitment, denoted by $[x_i]$. The corresponding tag t_i held by the prover we sometimes refer to as a MAC on x_i, denote by $M[x_i]$, with the key β_i held by the verifier, which we denote by $K[x_i]$.

The scheme enjoys the usual properties of *hiding* and *binding*: For the first property to hold x_i must remain secret to the verifier until the commitment is opened, which is immediate from security of the VOLE, since any secure VOLE protocol must ensure that x_i is not leaked to the sender. For the second property to hold the prover must not be able to open up the commitment to a any other value than x_i. It's not hard too see that the prover's best chance at doing this is by guessing Δ, that is, if t is an attempted opening to a different value $x \neq x_i$ and the verifier accepts this opening, then

$$t_i = \Delta x_i + \beta_i \quad \text{and} \quad t = \Delta x + \beta_i \tag{3.1}$$

so the prover can compute $(x - x_i)^{-1}(t - t_i) = \Delta$. However, Δ is a random field element only known to the verifier, so the probability of the prover doing this is $1/|\mathbb{F}|$.

The commitment scheme also enjoys another delightful property, namely it is an additively *homomorphic* commitment scheme. It straight forward to see this, since if t_i, t_j are commitments to x_i, x_j with keys β_i, β_j respectively then

$$t_i + t_j = \Delta x_i + \beta_i + \Delta x_j + \beta_j = \Delta(x_i + x_j) + \beta_i + \beta_j$$

so $t_i + t_j$ is a commitment to $x_i + x_j$ with key $\beta_i + \beta_j$.

This property of VOLE commitments is essential to how they are used in the ZK proof, which will be clear is section 3.3 when I explain how the QuickSilver multiplication checks works.

3.2.1 VOLE in $\mathbb{Z}_{2^k}$

In the argument I gave using equation 3.1, for why the prover cannot break binding with probability greater than $1/|\mathbb{F}|$, it is essential that the underlying algebraic structure used to realize VOLE is a field. If we instead only require that the elements are from a ring R, then $(x - x_i)^{-1}$ may not exist (or phrased differently $x - x_i$ may be a zero divisor), and the security guarantees are no longer the same. In fact, in some rings, we can show that with probability $1/2$, then the prover can open a commitment to a different value than the one committed to and have the verifier accept this opening:

Consider $R = \mathbb{Z}_{2^k}$, which is a ring but not a field, since any even number is a zero-divisor[5]. Now, if Δ is chosen as $2\Delta'$ for some $\Delta' \in R$, which happens with probability $1/2$ if Δ is random, then the prover can open its commitment to $x + 2^{k-1}$ with the same tag t, effectively flipping the most significant bit in x. This will be accepted by the verifier since

$$\Delta(x + 2^{k-1}) = \Delta x + \Delta 2^{k-1} = \Delta x + 2\Delta' 2^{k-1} = \Delta x + \Delta' 2^k = \Delta x$$

where the last equality holds as we're working modulo 2^k.

The issue is that the VOLE-based commitment scheme is used to commit secrets $x \in R$, such that also $t, \Delta, \beta \in R$. To solve this problem, recent papers [BBMHS22] [BBMH+21] take an approach similar the information-theoretic MAC scheme of SPD$\mathbb{Z}_{2^k}$ [CDE+18], where commitments are from a larger ring than that of the values.

The idea is to instantiate random VOLE in an extension ring $\mathbb{Z}_{2^l}$ of $\mathbb{Z}_{2^k}$, where $l = k + s$ for some statistical security parameter s. In the setup phase the verifier learns the global key $\Delta \in_R \mathbb{Z}_{2^s}$, and when extending VOLE with a single correlation the prover learns random $r, M[r] \in_R \mathbb{Z}_{2^l}$, and the verifier learns $K[r] \in \mathbb{Z}_{2^l}$ subject to $M[r] = \Delta r + K[r] \bmod 2^l$ as usual. When the prover wants to commit to some $x \in \mathbb{Z}_{2^k}$ it will send $\delta = x - r \bmod 2^l$ and the verifier sets $M[x] = M[r] - \Delta\delta \bmod 2^l$ exactly as before.

[5]for $k > 1$, that is. $\mathbb{Z}_2$ is the prime field where addition and multiplication behaves like logical XOR and AND respectively.

Now, after performing linear operations locally on commitments, the prover will obtain some representative $\tilde{x} \in \mathbb{Z}_{2^l}$ for the committed value $x \in \mathbb{Z}_{2^k}$, i.e $\tilde{x} \bmod 2^k = x$. It is tempting to open commitments in the same way as before, by having the prover send $\tilde{x}, t$ and the verifier checking the linear constraint, however doing so could reveal more information than intended! If x is the result of adding commitments to y and z, then the upper s bits of the opening may reveal information on y and z (for instance if $y+z \geq 2^k$, i.e the operation overflowed). Consequently, if we were to open commitments in this naive way in the zero-knowledge proof, then we wouldn't be able to prove that the proof of knowledge is zero-knowledge. Thus, when opening a commitment the prover will use an extra random VOLE correlation to "blind" the upper s bits of the opening.

For this commitment-scheme it can be shown that any prover who can open a commitment to some other value than x, must be have guessed the upper s bits of Δ, which happens with probability 2^{-s} [CDE+18].

The following functionality from [BBMHS22] formalizes VOLE in $\mathbb{Z}_{2^k}$ as described above[6]:

Functionality $F^{l,s}_{VOLE2k}$: **VOLE in** $\mathbb{Z}_{2^k}$

Let $l \geq s$.

Init This method is the first to be called by both parties (besides CORRUPT). On input (INIT) from both parties:

1. If $\mathcal{V}$ is honest sample $\Delta \in_R \mathbb{Z}_{2^s}$ and send Δ to $\mathcal{V}$.
2. If $\mathcal{V}$ is corrupt receive $\Delta \in \mathbb{Z}_{2^s}$ from $\mathcal{A}$.
3. Δ is stored by the functionality and all other (INIT) requests are ignored.

Extend On input (EXTEND, n) from both parties:

1. If $\mathcal{V}$ is honest sample $K[\vec{r}] \in_R \mathbb{Z}^n_{2^l}$, otherwise receive $K[\vec{r}] \in \mathbb{Z}^n_{2^l}$ from $\mathcal{A}$.
2. If $\mathcal{P}$ is honest sample $\vec{r} \in_R \mathbb{Z}^n_{2^k}$ and compute $M[\vec{r}] = \Delta \cdot \vec{r} + K[\vec{r}]$.
3. If $\mathcal{P}$ is corrupt receive $\vec{r} \in \mathbb{Z}^n_{2^k}$ and $M[\vec{r}] \in \mathbb{Z}^n_{2^l}$ and then recompute $K[\vec{r}] = K[\vec{r}] - \Delta \cdot \vec{r}$.
4. Send $(M[\vec{r}], \vec{r})$ to $\mathcal{P}$ and $K[\vec{r}]$ to $\mathcal{V}$.

Corrupt: On input (CORRUPT, X) from $\mathcal{A}$ with $X \in \{\mathcal{P}, \mathcal{V}\}$ corrupt X.

For my purpose of ZK, then [BBMHS22] shows that to achieve an expected probability of 2^{-s}, with which a cheating prover can convince the verifier that a circuit $C \notin ECSAT$ is satisfiable, I will need to choose k, l and s so that $l \approx k + 2(s + \log s)$. Since $k = 32$ is fixed by the word-size of

[6]The functionality in [BBMHS22] also exposes a method whereby a corrupt prover can query the global key Δ, which is needed when proving security of the VOLE protocol given in the same paper. This is not important in my setting, as I'll be relying on a trusted VOLE dealer, so I omit it.

MiniRAM, I've chosen $l = 128$ for about $s \approx 40$ bits of security. Then $x \in \mathbb{Z}_{2^l}$ can conveniently be represented using 128-bit unsigned integers in Rust, and the group and ring operations in $\mathbb{Z}_{2^l}$ conveniently coincide with wrapping unsigned integer addition and multiplication respectively.

3.2.2 Recent efficient VOLE generators

The efficient generation of large VOLE correlations is an active area of research. My implementation makes use of a trusted VOLE dealer, as I've considered its implementation as a secure two-party protocol out of scope, but this section gives an overview of recent techniques for VOLE protocols, that can be (and some have been [BCG+19]) efficiently implemented on modern hardware.

At a high level, the current state-of-the-art approach builds on an idea contributed by Boyle et. al. in [BCGI19], where a random VOLE *generator* is used to locally expand a pair of correlated short strings (or seeds) into a pseudorandom VOLE correlation. Since the expansion of the seeds happens locally, then this leads to VOLE constructions with communication much less than the output length, so this approach is also referred as *silent preprocessing* when used to do secure computations like ZK [BCG+19]. The technique involves combining function secret sharing (FSS) and noisy linear encodings in a clever way:

Given a class of functions $\mathcal{F} = \{f : I \to G\}$ with domain I and co-domain a commutative group G, then function secret sharing scheme for $\mathcal{F}$ consists of algorithms $Gen(f, \lambda \in \mathbb{N}) \to (K_0, K_1)$ and $Eval(b \in \{0,1\}, K_b, x \in I) \to y_b$, where, loosely speaking, the scheme is said to be correct if $y_0 + y_1 = f(x)$ for all x, and secure (or private) if K_b reveals no more information about f than its domain and co-domain.

A specific kind of FSS schemes are distributed point functions (DPF), where functions in $\mathcal{F}$ are all of the form $f_{\alpha,\beta} : I \to G$ where $f_{\alpha,\beta}(x) = \beta$ if $x = \alpha$ and 0 otherwise. They can be efficiently constructed from any pseudorandom generator such as AES [BGI18].

DPF generalizes to multi-point functions in the natural way, so yet another kind of FSS schemes are multi point function sharing scheme (MPFSS), where $\mathcal{F}$ consists of (n,t) multi-point functions of the form $f_{\vec{\alpha},\vec{\beta}} : I \to G$ for $\vec{\alpha} \in I_n^t$ and $\vec{\beta} \in G^t$, where $f_{\vec{\alpha},\vec{\beta}}(x) = \vec{\beta}_i$ if $x = \vec{\alpha}_i$ and 0 otherwise. They can be constructed naively using t DPF constructions for the single-point functions that $f_{\vec{\alpha},\vec{\beta}}$ is composed of (secure MPFSS schemes are also allowed to leak t from K_b).

Now, the observation made in [BCGI19] is that using a MPFSS scheme, then a "sparse" VOLE correlation can be constructed from a short seed, meaning a correlation where most of the entries are 0. To turn this sparse VOLE into a pseudorandom VOLE, the learning parity with noise (LPN) assumption is used, to add random code-words in a way that maintains the VOLE correlation.

3.3 QuarkSilver+

QuarkSilver+ is a specialization of QuarkSilver [BBMHS22] to support the special gates of extended arithmetic circuits over $\mathbb{Z}_{2^{32}}$, and QuarkSilver itself is an adaption QuickSilver [YSWW21] to support arithmetic circuits over rings such as $\mathbb{Z}_{2^k}$. All of the protocols follow the established commit-and-prove paradigm [CD96], where the prover commits to the witness, as well as the output of all non-linear gates, and shows that the input to non-linear gates are consistent with the committed output.

I'll begin this section, by elaborating on the intuition behind the commit-and-prove paradigm, and continue to explain how the multiplication checks from QuickSilver, which are central to QuarkSilver, works. I'll then give a high-level overview of the protocol QuarkSilver+ before, finally, giving the details of the protocol in all its glory.

Commit-and-prove ZK proofs using homomorphic commitments Imagine you're given a circuit C over $\mathbb{F}$ with n inputs, which consists solely of constant and addition gates, and Alice claims $C \in ECSAT$, and in fact, she *knows* $\vec{x} \in \mathbb{F}^n$ so that $C(\vec{x}) = 0$. This example is, of course, silly, since any linearly bounded verifier in the circuit size could easily apply simple backwards reasoning to figure out some $\vec{x}$ satisfying C, if C is not a constant non-zero circuit, but consider for the sake of this demonstration, that you want to be convinced that $C \in ECSAT$ by talking to Alice, instead of simply outputting TRUE.

Having access to an implementation of VOLE over $\mathbb{F}$, the following protocol is a complete ZK proof system, which follows directly from the fact that VOLE commitments are homomorphic.

1. Alice and you initialize the VOLE. You obtain $\Delta \in \mathbb{F}$.
2. Alice and you extend the VOLE correlation, obtaining n random VOLE correlations.
3. Alice and you run the subprotocol to commit to Alice's witness. You obtain $K[x_i]$ for $i \in [n]$.
4. Let $\vec{M}$ be the vector of tags, i.e $\vec{M}_i = M[x_i]$. Alice evaluates $C(\vec{M})$ obtaining MAC $M[o_i]$ for the input o_i of each output gate.
5. Let $\vec{K}$ be the vector of keys, i.e $\vec{K}_i = K[x_i]$. You evaluate $C(\vec{K})$ obtaining a MAC $K[o_i]$ for the input o_i of each output gate.
6. Alice sends you $M[o_i]$ for each output gate and you output TRUE if $M[o_i] = K[o_i]$ (asserting that $o_i = 0$) for all i and FALSE otherwise.

If Alice can convince you that a circuit that she doesn't know a satisfying witness for is satisfiable, then she must open the last commitment to $0 \neq o_i$, which she can only do with probability $1/|\mathbb{F}|$, by breaking binding and guessing Δ.

Now, what happens if the circuit contains a multiplication gate? Since commitments aren't fully homomorphic, it's not always the case that $M[x]M[y] = M[xy]$ or $K[x]K[y] = K[xy]$, so you, the verifier's, circuit evalution is seemingly stuck.

To overcome this problem, protocols in the commit-and-prove paradigm using additively homomorphic commitment schemes do the following: Assume the circuit has t multiplication gates and $l \in \mathbb{N}$ (I will return to the meaning of l in a second). In step 2 the VOLE is extended with $n + t + l$ instead of n. In step 4 Alice will also evaluate $C(\vec{x})$, and when evaluating a multiplication gate with inputs x, y, Alice will commit to xy (so you obtain $K[xy]$, and can continue the circuit evaluation in step 5.

Now a different problem arises: When committing to xy Alice might cheat! Therefore, a check to ensure consistency of $K[xy]$, $K[x]$ and $K[y]$ for multiplication gates must be added to the protocol, for it to be sound. This is where the extra l random VOLE correlations are used, and different strategies yield different bounds on l with different security guarantees.

3.3.1 QuickSilver multiplication checks

The fundamental idea, when consistency-checking multiplication gates in QuarkSilver, originated in [YSWW21], and I'll repeat it here. As stated above, the problem is this: Given values x, y, z from some field $\mathbb{F}$, and VOLE commitments to these, such that the prover holds $x, y, z, M[x], M[y], M[z] \in \mathbb{F}$ and the verifier holds $K[x], K[y], K[z], \Delta \in \mathbb{F}$, subject to $M[i] = \Delta i + K[i]$ for $i \in \{x, y, z\}$, a protocol to check if $xy = z$ is needed[7]. The crucial observation is that

$$\begin{aligned} K[x]K[y] + \Delta K[z] &= (M[x] - \Delta x)(M[y] - \Delta y) + \Delta(M[z] - \Delta z) \\ &= M[x]M[y] - \Delta x \cdot M[y] - M[x]\Delta y + \Delta^2 xy + \Delta M[z] - \Delta^2 z \\ &= M[x]M[y] - \Delta(M[y]x + M[x]y - M[z]) + \Delta^2(xy - z) \end{aligned} \tag{3.2}$$

so if $xy = z$ then

$$\underbrace{K[x]K[y] + \Delta K[z]}_{B} = \underbrace{M[x]M[y]}_{A_0} - \Delta \underbrace{(M[y]x + M[x]y - M[z])}_{A_1} \tag{3.3}$$

which is a linear equation of the form $B = A_0 - \Delta A_1$ where the verifier knows $B, \Delta \in \mathbb{F}$ and the prover knows $A_0, A_1 \in \mathbb{F}$. An (insufficient) protocol for consistency-checking multiplication gates is therefore for the prover to send A_0, A_1 to the verifier, who then checks if $B = A_0 - \Delta A_1$. This is complete, as shown above, and it can be shown that it has a soundness error of $2/|\mathbb{F}|$, which relies on the fact that any degree-2 polynomial over a field has at most 2 roots. It does, however, not permit consistency-checking multiplications in zero-knowledge, since the verifier learns the product of two secrets held by the prover, and thus any attempt at simulating A_0 fails (and similarly for A_1).

The solution is to observe that 3.3 imposes a constraint on B, A_0 and A_1 identical to that of VOLE correlations, so using a single fresh correlation $M[r], K[r], r \in \mathbb{F}$ the multiplication check can be made zero knowledge like this: The prover sends $A_0^* = A_0 + M[r]$ and $A_1^* = A_1 + r$ to the verifier (which to the verifier looks like, and thus can be simulated as, random values), who checks if $B^* = A_0^* - \Delta A_1^*$ where $B^* = B + K[r]$.

Instead of consistency-checking each multiplication gate individually, multiplication checks for all t gates in a circuit can be batched together in the following insufficient protocol: For the i'th multiplication gate the parties hold $[x_i], [y_i][z_i]$. The prover computes $A_{0,i} = M[x_i]M[y_i]$ and $A_{1,i} = M[y_i]x_i + M[x_i]y_i - M[z_i]$ and the verifier computes $B_i = K[x_i]K[y_i] + \Delta K[z_i]$. At the end of circuit evaluation the prover sends

$$A_0^* = \sum_{i=1}^{t} A_{0,i} + M[r] \quad \text{and} \quad A_1^* = \sum_{i=1}^{t} A_{1,i} + r$$

to the verifier, who checks if $B^* = A_0^* - \Delta A_1^*$ where $B^* = \sum_{i=1}^{t} B_i + K[r]$.

This protocol for consistency-checking a batch of t multiplications at once is complete and zero-knowledge, but it does not achieve a desirable soundess error, because the check doesn't ensure that $B_i = A_{0,i} - \Delta A_{1,i}$ for all $i \in [t]$. To fix this, in QuickSilver the verifier samples a random

[7]Since, again as explained above, a corrupt prover might cheat and commit to $z \neq xy$.

challenge $\chi \in_R \mathbb{F}$ independent of all $A_{0,i}, A_{1,i}$ (that is, the verifier samples χ after the prover has committed to the output of all multiplication gates). Then the verifier sends χ to the prover who computes and sends

$$A_0^* = \sum_{i=1}^{t} \chi^i \cdot A_{0,i} + M[r] \quad \text{and} \quad A_1^* = \sum_{i=1}^{t} \chi^i \cdot A_{1,i} + r$$

back to the verifier, who then checks if $B^* = A_0^* - \Delta A_1^*$ where $B^* = \sum_{i=1}^{t} \chi^i B_i + K[r]$. It can be shown [YSWW21] that this protocol for consistency-checking the output of all multiplication gates achieves a soundness error of $(t+2)/|\mathbb{F}|$, which crucially relies on the fact that the verifier samples χ at random *after* having received $A_{0,i}, A_{1,i}$ for $i \in [t]$ from the prover.

3.3.2 QuarkSilver multiplication checks

To achieve the desired soundness error in the (batched) multiplication check of QuickSilver as described above, it is essential to that we're working in a field, where a degree n polynomial has at most n roots. Therefore, when committing to values in $\mathbb{Z}_{2^k}$ using the functionality $F_{\text{VOLE2k}}^{l,s}$ the protocol and security analysis needs to be adapted. This is (part of) the contributions of [BBMHS22], which adapts the batched multiplication consistency-checks the following way: Instead of the verifier sampling a random challenge $\chi \in_R \mathbb{Z}_{2^l}$, then now a random vector $\vec{\chi} \in_R \mathbb{Z}_{2^l}^t$ is sampled by the verifier, after the prover has sent deltas for committing to the output of all multiplication gates. The prover then computes

$$A_0^* = \sum_{i=1}^{t} \vec{\chi}[i] \cdot A_{0,i} + M[r] \bmod 2^l \quad \text{and} \quad A_1^* = \sum_{i=1}^{t} \vec{\chi}[i] \cdot A_{1,i} + r \bmod 2^l$$

and sends to the verifier, who performs the same check as before, except also using the i'th entry from $\vec{\chi}$ instead of the i'th power of a field element. Then [BBMHS22] performs a careful analysis of the number of roots modulo 2^l of a polynomial of the form $f(x) = ax^2 + bx + c$, in the case where $a \neq 0$ modulo 2^k (since the prover is cheating and committing to $z \neq xy$), bounding the soundness error in the batched consistency-checks with 2^{-s} for statistical security parameter $s \geq 7$, when $l, k \in \mathbb{N}$ satisfy

$$l = k + 2(s + \log(s) + 3) \approx k + 2(s + \log(s))$$

3.3.3 How QuarkSilver+ works

Before presenting the protocol in details, I will now give a high-level overview of how QuarkSilver+ works. In particular, this overview describes how the protocol works for the non-standard and non-linear gates **Select**, **SelectConst**, **Decode32** and **CheckAllEqButOne**.

The two parties must have access to a secure implementation of $F_{\text{VOLE}}^{128,48}$. Then, given a circuit C over $\mathbb{Z}_{2^{32}}$ with n input gates, a multiplication gates, b **Select** alternatives, c **SelectConst** alternatives, d **Decode32** gates, and e pairs of the **CheckAllEqButOne** gates, the parties proceed as follows:

- The prover and verifier pre-process $n + a + 2b + c + 32d + e + 1$ random VOLE correlations in $\mathbb{Z}_{2^{128}}$.

- The prover commits to the secret witness $w_1, ..., w_n$ (the input of the circuit), and obtains tags $M[w_1], ..., M[w_n]$.
- The prover evaluates the circuit on the secret witness $w_1, \dots, w_n$ as well as on the tags $M[w_1], \dots, M[w_n]$, where evaluation of all linear gates happens locally, due to the homomorphic property of VOLE commitments. Evaluation of non-linear gates on the tags happen as follows:
  - For each multiplication gate the prover commits to the output of the gate using 1 random VOLE correlation.
  - For each Select gate with inputs $i, x_0, ..., x_{m-1}$ the prover commits to
  $$b_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \quad \text{and } b_j x_j \text{ for } j \in [m-1] \tag{3.4}$$
  using $2m$ random VOLE correlations. The prover then convinces the verifier that all committed values satisfy 3.4 by showing that $[b_j]([b_j]-1) = [0]$, which shows that they are all bits, $[b_j]([i]-j) = [0]$ for all j (which shows that $[b_j] = [0]$ for all $j \neq i$) and $\sum_{j=0}^{m-1}[b_j] = [1]$ (which together with the previous fact implies $[b_i] = [1]$). Finally, the output of the gate is then computed as $\sum_{j=0}^{m-1}[x_j b_j] = [x_i]$.
  - For each SelectConst gate with inputs $i, c_0, ..., c_{m-1}$ the prover commits to
  $$b_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \quad \text{for } j \in [m-1] \tag{3.5}$$
  using m random VOLE correlations. The prover then convinces the verifier that all committed values satisfy 3.5, by showing that $[b_j]([b_j]-1) = [0]$ and $[b_j](i-j) = 0$ for all j, and that $\sum_{j=0}^{m-1}[b_j] = [1]$, similar to **Select** gates. The output of the gate is similarly computed as $\sum_{j=0}^{m-1} c_j[b_j] = [c_i]$.
  - For each Decode32 gate with input x the prover commits to
  $$x_j \in \{0,1\} \quad \text{for } j = 1, \dots, 32$$
  such that $\sum_{i=1}^{32} 2^{i-1}x_i = x$, using 32 random VOLE correlations. The prover then convinces the verifier that committed values are bits by showing that $[x_j](1-[x_j]) = 0$ for all j, and that the x_j's is a correct bit-decomposition of x by showing that $[x] - \sum_{i=1}^{32} 2^{i-1}[x_i] = [0]$. The (multi-valued) output of the gate is then computed as $[x_1], .., [x_{32}]$.
  - For each CheckAllEqButOne gate with inputs $i, x_0, y_0, x_1, y_1, \dots, x_{m-1}, y_{m-1}$ the prover commits to
  $$b_j = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if } i \neq j \end{cases} \quad \text{for } j = 0, \dots, m-1 \tag{3.6}$$
  using m random VOLE correlations. The prover then shows that the committed values satisfy 3.6 by showing that $[b_j]([b_j]-1) = [0]$ and $([b_j]-1)(i-j) = [0]$ for all j (proving

that $[b_j] = [1]$ for $i \neq j$) and that $\sum_{i=1}^{m}[b_j] = [m-1]$ (which together with the previous fact implies that $[b_i] = [0]$, similar to **Select** gates). Finally, the prover convinces the verifier that the assertion of the gate holds, that is $x_j = y_j$ for all $j \neq i$, by showing that $[b_j][x_j - y_j] = [0]$ for all j.

- Finally, the prover will show that the committed values for the non-linear gates are consistent with the inputs of the gate, using the multiplication checks from QuarkSilver, and that the (committed) value flowing into any output gate is 0.

Figure 3.4 gives the protocol in detail. It is a specialization of Quarksilver [BBMHS22] to my setting of extended arithmetic circuits over $\mathbb{Z}_{2^{32}}$, for the purpose of conveniently constructing circuit compilers for 32-bit architectures. It securely implements the functionality F_{ZK}^{k} for $k = 32$, the proof of which I'll give in the upcoming section.

Protocol: **QuarkSilver+**

The prover $\mathcal{P}$ and verifier $\mathcal{V}$ have agreed on a circuit C over $\mathbb{Z}_{2^k}$ with n **In** gates, a **Mul** gates, b' **Select** gates with a total of b alternatives, c' **SelectConst** gates with a total of c alternatives, d **Decode32** gates, e' **CheckAllEqButOne** gates with a total of e pairs, and f **Out** gates, and $\mathcal{P}$ holds a witness $\vec{w} \in \mathbb{Z}_{2^k}^{n}$ so that $C(\vec{w}) = 0$.

Preprocessing phase Let $\theta = a + 2b + c + 32d + e + f$. The preprocessing phase is independent of C and just needs an upper bound on $n + \theta + 1$.

1. $\mathcal{P}$ and $\mathcal{V}$ send (INIT) to $F_{\text{VOLE}}^{l,s}$, and $\mathcal{V}$ receives $\Delta \in_R \mathbb{Z}_{2^s}$.
2. $\mathcal{P}$ and $\mathcal{V}$ send (EXTEND, $n + \theta + 1$) to $F_{\text{VOLE}}^{l,s}$ which returns authenticated values $[r_i]$ to the parties for $i \in [n + \theta + 1]$, and where $\tilde{r}_i$ are random values from $\mathbb{Z}_{2^l}$.

Online phase For each gate in topological order:

- If the gate is an **In**$_i$ gate, then $\mathcal{P}$ sends $\delta_{r_i} = w_i - \tilde{r}_i$ to $\mathcal{V}$ and both parties locally compute $[w_i] = [r_i] + \delta_{r_i}$.
- If the gate is an **Add** gate with inputs $\tilde{x_1}, \ldots, \tilde{x_m} \in \mathbb{Z}_{2^l}$, then the parties hold $[x_i]$ for $i \in [m]$, and both parties compute $[\sum_{i=1}^{n} x_i] = \sum_{i=1}^{n}[x_i]$.
- If the gate is a **Sub** gate with inputs $\tilde{x}, \tilde{y} \in \mathbb{Z}_{2^l}$, then the parties hold $[x]$ and $[y]$, and both parties locally compute $[x - y] = [x] - [y]$.
- If the gate is a **Mul** gate with inputs $\tilde{x}, \tilde{y} \in \mathbb{Z}_{2^l}$, then the parties hold $[x]$ and $[y]$. Let $[r_i]$ be the next authenticated value. $\mathcal{P}$ then sends $\delta_{r_i} = xy - \tilde{r}_i$ to $\mathcal{V}$, and both locally compute $[xy] = [r_i] + \delta_{r_i}$ and run **CheckMul**([x],[y],[xy]).
- If the gate is a **Const** gate with parameter $c \in \mathbb{Z}_{2^k}$, then the parties locally compute $[c]$ by setting $M[c] = 0$ and $K[c] = -\Delta c$.

- If the gate is a **MulConst** gate with input $\tilde{x} \in \mathbb{Z}_{2^l}$ and parameter $c \in \mathbb{Z}_{2^k}$, then the parties hold $[x]$ and locally compute $[cx] = c \cdot [x]$.

- If the gate is a **Select** gate with inputs $\tilde{i}, \tilde{x}_0, ..., \tilde{x}_{m-1} \in \mathbb{Z}_{2^l}$ then the parties hold $[i]$ and $[x_0], ..., [x_{m-1}]$. Let $[r_0], [r_1], .., [r_{2m-1}]$ be the next $2m$ unused authenticated values. The parties proceed with the following steps:

  1. For each $j \in [m]$:
     (a) $\mathcal{P}$ computes
     $$b_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$
     and sends $\delta_{r_j} = b_j - r_j$ and $\delta_{r_{j+m}} = b_j x_j - r_{j+m}$ to $\mathcal{V}$.
     (b) They both locally compute $[b_j] = [r_j] + \delta_{r_j}, [b_j x_j] = [r_{j+m}] + \delta_{r_{j+m}}, [i - j] = [i] - j$ and $[b_j - 1] = [b_j] - 1$.
     (c) The parties run **CheckMul**$([b_j], [b_j - 1], [0])$, **CheckMul**$([b_j], [i - j], [0])$ and **CheckMul**$([b_j], [x_j], [b_j x_j])$.
  2. The parties locally compute $[b] = \sum_{j=0}^{m-1} [b_j]$ and run **Assert**$([b], 1)$.
  3. The parties locally compute $[x_i] = \sum_{j=0}^{m-1} [b_j x_j]$.

- If the gate is a **SelectConst** gate with input $\tilde{i} \in \mathbb{Z}_{2^l}$ and public parameters $c_0, ..., c_{m-1}$ in $\mathbb{Z}_{2^k}$ then the parties hold $[i]$. Let $[r_0], [r_1], .., [r_{m-1}]$ be the next m unused authenticated values. The parties then proceed like this:

  1. For each $j \in [m]$:
     (a) They locally compute $[c_j]$ by setting $M[c_j] = 0$ and $K[c_j] = -\Delta c_j$.
     (b) Similar to **Select** gates, $\mathcal{P}$ now computes
     $$b_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$
     but $\mathcal{P}$ only sends $\delta_{r_j} = b_j - r_j$ to $\mathcal{V}$.
     (c) The parties then locally compute $[b_j] = [r_j] + \delta_{r_j}, [b_j c_j] = [b_j] c_j, [i - j] = [i] - j$ and $[b_j - 1] = [b_j] - 1$.
     (d) The parties run **CheckMul**$([b_j], [b_j - 1], [0])$ and **CheckMul**$([b_j], [i - j], [0])$.
  2. The parties locally compute $[b] = \sum_{j=0}^{m-1} [b_j]$ and run **Assert**$([b], 1)$.
  3. The parties locally compute $[c_i] = \sum_{j=0}^{m-1} [b_j] c_j$.

- If the gate is an **Encode4**, **Encode8** or **Encode32** gate with inputs $\tilde{x}_1, ..., \tilde{x}_i \in \mathbb{Z}_{2^l}$ for $i = 4$ or $i = 8$ or $i = 32$, then the parties hold $[x_j]$ for $j \in [i]$, and both parties locally compute $[\sum_{j=1}^{i} 2^{j-1} x_j] = \sum_{j=1}^{i} 2^{j-1} [x_j]$.

- If the gate is a **Decode32** gate with input $\tilde{x} \in \mathbb{Z}_{2^l}$, where $\tilde{x} \bmod 2^{32} = \sum_{i=0}^{31} 2^i x_i$ for bits x_i, then the parties hold $[x] = [\sum_{i=0}^{31} 2^i x_i]$. Let $[r_0], [r_1], .., [r_{31}]$ be the next 32 unused authenticated values. The parties proceed as follows:
    1. For $i \in [32]$:
        (a) $\mathcal{P}$ sends $\delta_{r_i} = x_i - \tilde{r}_i$ to $\mathcal{V}$ and both parties locally compute $[x_i] = [r_i] + \delta_{r_i}$ and $[1 - x_i] = 1 - [x_i]$.
        (b) The parties run **CheckMul**$([x_i], [1 - x_i], [0])$.
    2. The parties locally compute $[\sum_{i=0}^{31} 2^i x_i] = \sum_{i=0}^{31} 2^i [x_i]$.
    3. The parties run **Assert**$([x] - [\sum_{i=0}^{31} 2^i x_i], 0)$.

- If the gate is an **Out** gate with input $\tilde{x}$, then the parties hold $[x]$ and run **Assert**$([x], 0)$.

- If the gate is a **CheckAllEqButOne** with inputs $\tilde{i}, \tilde{x}_0, \tilde{y}_0, ..., \tilde{x}_{m-1}, \tilde{y_{m-1}} \in \mathbb{Z}_{2^l}$, then the parties hold $[i], [x_0], [y_0], ..., [x_{m-1}], [y_{m-1}]$. Let $[r_0], [r_1], .., [r_{m-1}]$ be the next m unused authenticated values. The parties proceed like this:
    1. For each $j \in [m]$:
        (a) $\mathcal{P}$ computes
        $$b_j = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{otherwise} \end{cases}$$
        and sends $\delta_{r_j} = b_j - r_j$ to $\mathcal{V}$.
        (b) Both parties now locally compute $[b_j] = [r_j] + \delta_{r_j}, [b_j - 1] = [b_j] - 1, [i - j] = [i] - j, [x_j - y_j] = [x_j] - [y_j]$ and $[b_j - 1] = [b_j] - 1$.
        (c) They run **CheckMul**$([b_j], [b_j - 1], [0])$, **CheckMul**$([b_j], [x_j - y_j], [0])$ and **CheckMul**$([b_j - 1], [i - j], [0])$.
    2. The parties locally compute $[\sum_0^{m-1} b_j] = \sum_0^{m-1} [b_j] - 1$.
    3. The parties run **Assert**$([\sum_0^{m-1} b_j], m - 1)$.

Finally, $\mathcal{P}$ and $\mathcal{V}$ run the following check:

1. Set $A_0^* = M[\text{o}], A_1^* = \tilde{\text{o}}$ and $B^* = K[\text{o}]$ so that $B^* = A_0^* - \Delta A_1^*$.
2. Let $\theta' = b' + c' + d + 2e' + f$. $\mathcal{V}$ samples $\chi_i \in_R \mathbb{Z}_s$ for $i \in [\theta']$ and sends these to $\mathcal{P}$.
3. $\mathcal{P}$ then computes $U = \sum_{i=1}^{\theta'} \chi_i A_{0,i} + A_0^* \bmod 2^l$ and $V = \sum_{i=1}^{\theta'} \chi_i A_{1,i} + A_1^* \bmod 2^l$ and sends (U, V) to $\mathcal{V}$.
4. $\mathcal{V}$ computes $W = \sum_{i=1}^{\theta'} \chi_i B_i + B^* \bmod 2^l$ and checks that $W = U - V\Delta \in \mathbb{Z}_{2^l}$, in which case $\mathcal{V}$ accepts and otherwise rejects.

CheckMul$([x],[y],[z])$ If this is the i'th call to **CheckMul** then $\mathcal{P}$ computes the next $A_{0,i} = M[x]M[y] \bmod 2^l$ and $A_{1,i} = M[y]\tilde{x} + M[x]\tilde{y} - M[z] \bmod 2^l$ and $\mathcal{V}$ computes the next $B_i = K[x]K[y] + \Delta K[z] \bmod 2^l$.

Assert$([x], y \in \mathbb{Z}_{2^k})$ Let $[r_i]$ be the next unused authenticated value. $\mathcal{P}$ computes $x' = \tilde{x} \gg 32$ and sends $\delta_{r_i} = x' - r_i$ to $\mathcal{V}$. Both parties then locally computes $[x'] = [r_i] + \delta_{r_i}, [2^{32}x'] = 2^{32}[x'], [1]$ and $[y]$ and then run **CheckMul**$([x] - [2^{32}x'], [1], [y])$.

Figure 3.4: Zero-knowledge protocol for extended arithmetic circuit satisfiability in $\mathbb{Z}_{2^k}$, in the $F^{l,s}_{\text{VOLE}}$-hybrid model where $k \leq 32$.

It is tempting to try and optimize the protocol by rewriting it to prove inner products, which can be proven at a smaller cost [YSWW21] [BBMHS22]. For instance, for **Select** gates if we define the function $f(x_0,...,x_{m-1}) = \sum_{j=0}^{m-1} x_i(i-j) - 1$ and prove that $f(b_0,..,b_{m-1}) = 0$, then it seems that they can be proven at an optimized cost of $\mathcal{O}(1)$ random VOLE correlations, including the cost of committing to the witness. However, this choice of f is obviously insufficient to show that $b_j = 0$ for $j \neq i$ and $b_i = 1$: Indeed $f(b_0,..,b_{m-1}) = 0$ if the b_j's satisfy the constraints, but if $i < m-1$ and $b_{i-1} = b_{i+1} = 1$ then $b_{i-1}(i-(i-1)) = 1$ and $b_{i+1}(i-(i+1)) = -1$, so the two terms cancel out in the inner product. It's not clear to me if a suitable degree-2 function exits that will guarantee the constraints in this case.

The sub-protocol **Assert** blinds the most significant $l-k$ (96) bits of the value $\tilde{x}$ in $[x]$, and then uses **CheckMul** to verify that $x \bmod 32 = y$. If instead I'd opened commitment the usual way (with blinding), by having the prover send $\tilde{z} = \tilde{x} + \tilde{r}_i 2^k$ and $M[z] = M[x] + M[r_i]2^k$ to $\mathcal{V}$ who checks that $\tilde{z} \bmod 2^k = y$ and $M[z] = \Delta z + K[x] + K[r_i]2^k$, then I'll run into issues in the security proof: In [YSWW21] and [BBMHS22] then only the last wire of the circuit is opened, and this opening contributes respectively $1/|\mathbb{F}|$ and 2^{-s} (by the union bound) to the probability, with which a corrupt prover can break soundness. Thus opening commitments directly in **Assert** would make the probability with which soundness can be broken depend on the circuit, specifically number of gates that uses **Assert**, which is not desirable.

It's not immediately clear if a corrupt prover can compute some $x' \neq \tilde{x} \gg 32$ and pass the multiplication check in **Assert**, even though $x \neq y$. However, the following theorem shows that for any $r \neq \tilde{x} \gg 32$, if the most significant $l-k$ bits in $\tilde{x} - 2^k r$ are all 0, which they must be to pass the multiplication check since $y \in \mathbb{Z}_{2^k}$, then in fact the prover behaved honestly and $r = \tilde{x} \gg 32$.

theorem: **Soundness of Assert**

Let $l,k,s \in \mathbb{N}^+$ where $l = k+s$, and let $x \in \mathbb{Z}_{2^k}, r,z \in \mathbb{Z}_{2^s}$ and $\tilde{x} = 2^k z + x \bmod 2^l$.
If $\tilde{x} - 2^k r \bmod 2^l < 2^k$ then $r = z$.

Proof It suffices to show that if $r \neq z$ then $\tilde{x} - 2^k r \bmod 2^l \geq 2^k$. By definition of $\tilde{x}$ then $\tilde{x} - 2^k r \bmod 2^l = 2^k(z-r) + x \bmod 2^l$, so when $r < z$ then $2^k(z-r)+x \bmod 2^l \geq 2^k$. Notice that

$$\begin{aligned}\tilde{x} - 2^k r \bmod 2^l &= -(2^k r - \tilde{x}) \bmod 2^l \\ &= 2^l - (2^k r - \tilde{x}) \bmod 2^l\end{aligned}$$

and when $r > z$ then $0 < 2^k r - \tilde{x} < 2^l$ so

$$\begin{aligned}\tilde{x} - 2^k r \bmod 2^l &= 2^l - (2^k r - \tilde{x}) \\ &= 2^{k+s} - (2^k r - 2^k z - x) \\ &= 2^k 2^s - 2^k r + 2^k z + x \\ &= 2^k(2^s - r + z) + x \\ &\geq 2^k\end{aligned}$$

where the last inequality holds since $r < 2^s$. $\square$

3.3.4 Security of QuarkSilver+

Like [BBMHS22] then the protocol is proven secure in the $F^{l,s}_{\text{VOLE}}$-hybrid model, meaning that the simulator is allowed to simulate the communication with $F^{l,s}_{\text{VOLE}}$. We specialize the security theorem and proof to our case when $k = 32, l = 128$ and $s = 48$, but the same statement is in principle true for any $k < 32, s = \sigma + \log(\sigma) + 3$ and $l = k + 2s$ for statistical security parameter σ[8].

Theorem: **Security of QuarkSilver+**

QuarkSilver+ securely realizes the functionality F^k_{ZK} for $k = 32$ in the $F^{l,s}_{\text{VOLE}}$-hybrid model instantiated with $l = 128$ and $s = 48$: No unbounded environment can distinguish the real execution from the ideal except with probability $\approx 2^{-40}$.

The proof of security is divided in two cases:

1. The case of the corrupted prover, where we need to give a simulator for proving *soundness* (if $C \notin ECSAT$ then the honest verifier rejects) and *knowledge extraction* (if the honest verifier accepts, then we can efficiently extract the witness from the prover).
2. The case of the corrupted verifier, where we need to give a simulator for proving *zero-knowledge* (an honest prover will never reveal more than $C \in ECSAT$).

In each case, a simulator $\mathcal{S}$ is constructed, that is given access to F^{32}_{ZK} and rewindable black-box access to the adversary $\mathcal{A}$, while it's emulating the functionality $F^{128,48}_{\text{VOLE}}$. The parties are assumed to have agreed on a public circuit over $\mathbb{Z}_{2^{32}}$ with n inputs, a multiplication gates, b **Select** alternatives, c **SelectConst** alternatives, d **Decode32** gates and e pairs of the **CheckAllEqButOne** gates.

The full proof of security is included in appendix A.

[8]If $k > 32$ then **Decode32** gates no longer perform the intended operation. This may not really be a problem for the security proof, but the protocol with these parameters is useless.

Chapter 4

MiniRAM

Programming circuits is tedious and exotic to most programmers, who are used to expressing algorithms in a high-level programming language, which is then compiled to the instruction set architecture (ISA) of a particular CPU.

In this chapter I describe MiniRAM, a reduced instruction set computer (RISC) inspired by TinyRAM [BSCG$^+$13], designed with the goal of enabling efficient compilation of MiniRAM programs to arithmetic circuits. MiniRAM can be used to express general algorithms and computations as programs written in an assembly language, whose "executions can be proven" in zero knowledge. The intuitive meaning of "proving an execution" of a MiniRAM program, is to prove that the program is *satisfiable*, in the same sense as captured by the decision problem *ECSAT* for circuits, that is, that there exists an input that will make the program return 0. I will formalize this notion of satisfiability of MiniRAM programs in section 4.1.1.

Like TinyRAM, then MiniRAM follows a Harvard architecture, where code and data are stored in disjoint memories, which is opposed to machines of today who follow the Von Neumann architecture. This has the consequence that programs cannot be manipulated as data, so programs cannot make use of run-time code generation techniques such as just-in-time (JIT) compilation. [BSCTV14] shows how to adapt the approach to work for a Von Neumann architecture.

I will refer to these separate memories as "program-memory" and "data-memory", when the distinction is needed, or sometimes just "memory", when it's clear which one I mean. I will begin this chapter by giving the syntax and semantics of MiniRAM.

Unlike TinyRAM, whose compiler targets arithmetic circuits over a field, then the TinyRAM compiler targets extended arithmetic circuits over $\mathbb{Z}_{2^{32}}$ as presented in the last chapter. This makes the internals of the circuit compiler a little bit simpler, since I'm leveraging the specialized gates **Select**, **Decode32**, **Encode32**, **Encode8** and **CheckAllEqButOne** heavily, for instance for instruction fetch, instruction decoding, routing the values of registers to the ALU, and consistency-checking the values of all registers. In section 4.2 I will explain the details of the MiniRAM compiler.

4.1 Syntax and semantics

The syntax of MiniRAM is given in figure 4.1. The machine has 16 registers, the first holding the program counter, and they all hold 32-bit words. Instructions have a format similar to those of

Word	$w \in$	$\{0,\dots,2^{32}-1\}$	
Register	$r \in$	$\{pc, p_1, \dots, r_{15}\}$	
Instruction	$i ::=$	$\bowtie r\ r\ r$	$\bowtie \in \{\textbf{add}, \textbf{sub}\}$
	$\mid$	$\textbf{mov}\, r\ [r \mid w]$	move word from register or constant into register
	$\mid$	$\textbf{ldr}\, r\ r$	load word from memory into register
	$\mid$	$\textbf{str}\, r\ r$	store word from register into memory
	$\mid$	$\textbf{b}\, c^{?}\, r$	branching (conditional and unconditional)
	$\mid$	$\textbf{ret}\, [r \mid w]$	return word from register or constant
Condition	$c ::=$	Z	set if the last instruction was 0
Program	$p ::=$	$[i_0, i_1, \dots, i_n]$	where $n < 2^{32}$

Figure 4.1: Syntax of MiniRAM

ARM, where the left-most operand is the destination register. The machine supports moving words or constants between registers, basic arithmetic operations on words in registers (only wrapping addition and subtraction, that is, the usual group operation in $\mathbb{Z}_{2^{32}}$), control flow by means of conditional and unconditional branching, as well as basic word-aligned memory operations.

A program p is a sequence instructions, and I will use the notation $p(n)$ for $n \in \mathbb{N}$ to refer to the (0-indexed) n'th instruction in p, which is undefined if $n \geq 2^{32}$, since the program counter can't refer to any instructions above this.

Control flow The machine supports control flow with the instructions **b** for conditional branching and **bc** for unconditional branching. The operand to **b** and **bc** is a register, which holds the address of an instruction (from program-memory) to execute next.

```
1    mov r1 pc
2    mov r2 0x200100000000002a ; encoding of 'ret 42'
3    str r1 r2
4    b r1
```

Figure 4.2: Program and data are stored in separate memories, so instructions for branching and reading from/ writing to memory operate on different memories. The above program returns 42 on a Von Neumann architecture, but loops infinitely on MiniRAM's Harvard architecture.

Figure 4.2 illustrates how branching and memory instructions operate on two different memories: The program stores the encoding[1] of **ret** 42 at the address of the program counter in line 1, and then jumps to this address on line 4. On a Von Neumann architecture then this would have the effect of

[1]The encoding of instructions is explained in section 4.2.1

jumping to the newly stored instruction, returning 42. However, in MiniRAM, since the **str** at line 3 writes to a different memory than the **b** in line 4 reads from, then this has the effect of looping back to line 1.

Semantics The semantics of MiniRAM is given in figure 4.3 as a small-step operational semantics, which describes the individual steps of computation using a *transition relation*. This relation is parameterized by a program p, and relates 3-tuples of a register store s, which maps registers to their values, a memory heap h, which maps locations to their values, and the binary value of all conditional flags κ [2]. I will refer to such a 3-tuple (s,h,κ) as a *configuration*, since it completely describes the state of the machine, with respect to some program p.

Two configurations (s,h,κ) and (s',h',κ') are related by the transition relation, with respect to a program p, if looking up the value of the program counter in s yields a word w, and reading and decoding the w'th instruction from p yields i, and, intuitively, taking a single step according to i yields (s',h',κ'). This notion of taking a single step according to some instruction is formalized by the *step relation*, which again is parameterized by p, and relates a configuration *as well as an instruction* with another configuration. The step relation makes use of subroutines $\mathit{incPc} : \text{Store} \to \text{Store}$, for incrementing the machine's program counter, and $\mathit{isZero} : \text{Words} \to \{0,1\}$ for setting the value of the zero flag.

The defining rules of the step relation are rather straight forward. They capture the semantics of MiniRAM programs similar to how one would implement an interpreter for the language. For instance, the first rule relates an arithmetic instruction $\bowtie r_{dst}\ r_{lhs}\ r_{rhs}$ and configuration (s,h,κ) to the configuration (s',h,κ'), where the store s has been updated so the register r_{dst} now maps to the result of applying the mathematical operator corresponding to $\bowtie$, to whatever values s maps r_{lhs} and r_{rhs} to, and the value of the zero flag Z is appropriately set in κ'.

I have chosen to update the value of the conditional flag for all instructions except for writes to memory, in order to simplify the circuit for consistency checking the conditional flag, which I'll give more details on in section 4.2.2.

Another, perhaps unconventional, choice of mine, is to relate a return instruction i and configuration (s,h,κ) to the configuration (s',h,κ') where s' has been updated so the register r_1 contains the resulting value (which for the bottom-left rule is a word, and the bottom-right rule is from a register). Since the program counter is not incremented, then this rule applies again, relating i and (s',h,κ') to (s',h,κ'), which effectively means, that when the machine returns it puts the return value in r_1 and enters an infinite loop. I found this convenient when defining satisfiability of MiniRAM programs, as explained in the next section, but it's not strictly necessary.

4.1.1 Defining MiniRAM satisfiability

Having formally defined the semantics of MiniRAM with a small-step operational semantics, I can now finally define what it means for a MiniRAM program to be *satisfiable*. Intuitively, the meaning is exactly the same as with arithmetic circuits, but with one key difference: Because evaluation of MiniRAM programs may diverge in the presence of infinite loops, whereas evaluation of circuits is

[2] MiniRAM currently just has one conditional flag - the zero flag Z - so using a singleton map to maintain the value of Z is arguably a bit overkill, however this formalization allows for easily extending the machine with more flags.

Store	s	$\in$	Register $\to$ Word
Address	l	$\in$	Word $2^{32}-1$
Heap	h	$\in$	Address $\to$ Word
Flags	κ	$\in$	$\{\mathrm{Z}\} \to \{0,1\}$

Transition relation

$$\frac{s(pc) = w \quad \mathrm{decode}(p(w)) = i \quad \langle i, s, h, \kappa\rangle \to_p \langle s', h', \kappa'\rangle}{\langle s, h, \kappa\rangle \to_p \langle s', h', \kappa'\rangle}$$

Step relation

$$\frac{w = s(r_{lhs}) \bowtie s(r_{rhs}) \quad s^* = s[r_{dst} \mapsto w] \quad s' = \mathrm{incPc}(s^*) \quad \kappa' = [\mathrm{Z} \mapsto \mathit{isZero}(w)]}{\langle \bowtie r_{dst}\ r_{lhs}\ r_{rhs}, s, h, \kappa\rangle \to_p \langle s', h, \kappa'\rangle} \qquad \bowtie \in \{\mathbf{add}, \mathbf{sub}\}$$

$$\frac{s^* = s[r_{dst} \mapsto s(r_{src})] \quad s' = \mathrm{incPc}(s^*) \quad \kappa' = [\mathrm{Z} \mapsto \mathit{isZero}(s(r_{src}))]}{\langle \mathbf{mov}\ r_{dst}\ r_{src}, s, h, \kappa\rangle \to_p \langle s', h, \kappa'\rangle}$$

$$\frac{s^* = s[r_{dst} \mapsto w] \quad s' = \mathrm{incPc}(s^*) \quad \kappa' = [\mathrm{Z} \mapsto \mathit{isZero}(w)]}{\langle \mathbf{mov}\ r_{dst}\ w, s, h, \kappa\rangle \to_p \langle s', h, \kappa'\rangle}$$

$$\frac{s(r_{src}) = l \quad s^* = s[r_{dst} \mapsto h(l)] \quad s' = \mathit{incPc}(s^*) \quad \kappa' = [\mathrm{Z} \mapsto \mathit{isZero}(h(l))]}{\langle \mathbf{ldr}\ r_{dst}\ r_{src}, s, h, \kappa\rangle \to_p \langle s', h, \kappa'\rangle}$$

$$\frac{s(r_{dst}) = l \quad h' = h[l \mapsto s(r_{src})] \quad s' = \mathit{incPc}(s)}{\langle \mathbf{str}\ r_{dst}\ r_{src}, s, h, \kappa\rangle \to_p \langle s', h', \kappa\rangle}$$

$$\frac{s' = s[pc \mapsto p(r)] \quad \kappa' = [\mathrm{Z} \mapsto \mathit{isZero}(p(r))]}{\langle \mathbf{b}\ r, s, h, \kappa\rangle \to_p \langle s', h, \kappa'\rangle} \qquad \frac{\kappa(\mathrm{Z}) = 0 \quad s' = \mathit{incPc}(s)}{\langle \mathbf{b}\,\mathrm{Z}\ r, s, h, \kappa\rangle \to_p \langle s', h, \kappa\rangle}$$

$$\frac{\kappa(\mathrm{Z}) = 1 \quad s' = s[pc \mapsto p(r)] \quad \kappa' = [\mathrm{Z} \mapsto \mathit{isZero}(p(r))]}{\langle \mathbf{b}\,\mathrm{Z}\ r, s, h, \kappa\rangle \to_p \langle s', h, \kappa'\rangle}$$

$$\frac{\kappa' = [\mathrm{Z} \mapsto \mathit{isZero}(w)] \quad s' = s[r_1 \mapsto w]}{\langle \mathbf{ret}\ w, s, h, \kappa\rangle \to_p \langle s', h, \kappa'\rangle} \qquad \frac{\kappa' = [\mathrm{Z} \mapsto \mathit{isZero}(s(r))] \quad s' = s[r_1 \mapsto s(r)]}{\langle \mathbf{ret}\ r, s, h, \kappa\rangle \to_p \langle s', h, \kappa'\rangle}$$

Figure 4.3: Small-step operational semantics of MiniRAM

linearly bounded by the circuit size, then I will say that a MiniRAM program p is *t-satisfiable* if there exists an input x such that running p on x for *at most t steps* makes the machine return 0[3].

Inputs To define t-satisfiability of MiniRAM programs the first thing needed is a way to pass a sequence of words as input to the machine. I have chosen a strategy similar to [BSCG$^+$13], where the machine's memory is initialized with the program input laid out at consecutive addresses starting from address 0. Figure 4.4 shows how a program then read its arguments from memory when needed.

```
    mov r1 0
    ldr r1 r1   ; load first argument
    mov r2 1
    ldr r2 r2   ; load second argument
    add r1 r1 r2
    ret r1
```

Figure 4.4: Inputs are passed to MiniRAM programs, by initializing the machine's memory with the inputs laid out at consecutive addresses starting from address 0. The above program computes the function $f(x,y) = x + y$.

I define a logical predicate *Agree* for capturing when a heap h is initialized - or "agrees" - with some input $x \in \text{Word}^n$: $Agree(x,h)$ is true only if $h(i) = x_i$ for all $i \in \{0,..,n-1\}$. Notice that if $n-1 < 2^{32}$ then there are many heaps initialized with some $x \in \text{Word}^n$. The prover may initialize the machine with any of these, which means reading uninitialized memory returns non-deterministic values. I will return to how this is allowed by the circuit that checks memory consistency.

The final thing needed before I can state what it means for programs to be satisfiable is to construct, from the transition relation, a transitive relation $\xrightarrow{\textbf{ret}}{}^{t}_{p}$, that relates starting configurations (s,h,κ) to the "returning" configuration that the machine ends up in, when transitioning t times according to the transition relation. The relation is defined inductively by the following two rules:

$$\frac{s(pc) = w \quad decode(p(w)) = \textbf{ret}\, x}{\langle s,h,\kappa\rangle \xrightarrow{\textbf{ret}}{}^{0}_{p} \langle s,h,\kappa\rangle}$$

$$\frac{\exists s^*,h^*,\kappa^* : \quad \langle s,h,\kappa\rangle \rightarrow_p \langle s^*,h^*,\kappa^*\rangle \quad \langle s^*,h^*,\kappa^*\rangle \xrightarrow{\textbf{ret}}{}^{t-1}_{p} \langle s',h',\kappa'\rangle}{\langle s,h,\kappa\rangle \xrightarrow{\textbf{ret}}{}^{t}_{p} \langle s',h',\kappa'\rangle}$$

The first rule relates every returning configuration, that is, a configuration whose program counter is pointing to a **ret** instruction, to itself. The second rule says that from the starting configuration

[3]A more formal argument for why the notion of t-satisfiability is more sensible to use is the following: Had I defined satisfiability of programs similar to circuits, then, since MiniRAM is Turing complete, any proof system for proving satisfiability of MiniRAM programs could be used to solve the halting problem. But since the halting problem is undecidable such a system cannot exist.

(s,h,κ), if one can transition to some configuration (s^*,h^*,κ^*) using the transition relation, and subsequently step $t-1$ times to end up in the returning configuration (s',h',κ'), then, from (s,h,κ), one can step t times to end up in the returning configuration (s',h',κ').

Finally, we can say what it means for a MiniRAM program to be satisfiability within t steps of execution:

Definition: **MiniRAM satisfiability**

MRSAT is the following decision problem: Given a MiniRAM program p and time bound $t \in \mathbb{N}$, decide if there exists an input $x \in \text{Word}^n$, $s \in \text{Store}$, $\kappa \in \text{Flags}$ and $h,h' \in \text{Heap}$, such that

$$Agree(x,h) \quad \text{and} \quad \langle s_{init}, h, \kappa_{init}\rangle \overset{\mathbf{ret}}{\longrightarrow}{}_p^{\,t} \langle s,h',\kappa\rangle \quad \text{and} \quad s(r_1)=0$$

where s_{init} maps all registers to the word 0, and κ_{init} maps Z to the word 0.

I say that the program p is t-satisfiable if (p,t) is a yes-instance of *MRSAT*. If p is t satisfiable then p is of course also s-satisfiable for all $s > t$.

It should be clear how *MRSAT* formalizes the intuitive notion of t-satisfiability previously explained: When given some program p and time bound t, if we can decide that there exists an input x that agrees with some initial heap h, and the machine can transition t times to end up in a returning configuration (s,h',κ) where the value of register r_1, through which the machine returns, is 0, then the program is t-satisfiable.

Reducing *MRSAT* to *ECSAT* The job of the circuit compiler, which is explained in the next section, is to construct efficiently computable functions f_1 and f_2 such that for all programs p and time bounds $t \in \mathbb{N}$:

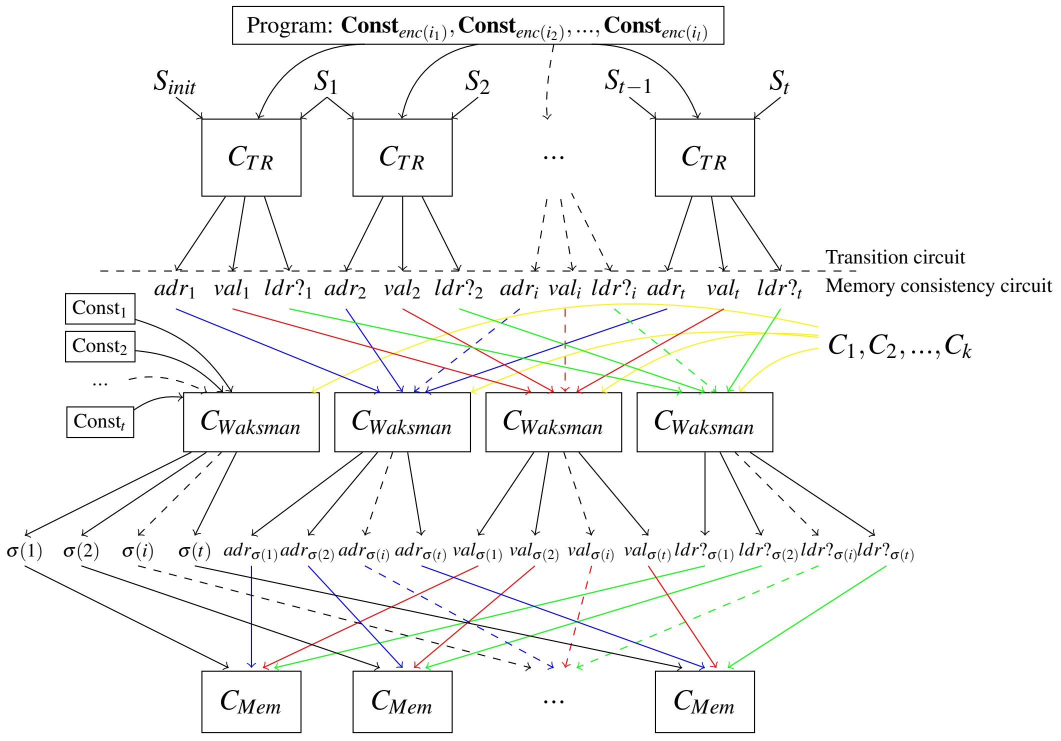
1. $(p,t) \in$*MRSAT* if and only if $f_1(p,t) \in$*ECSAT*, and
2. $\forall x \in \text{Word}^n : p_t(x) = 0$ if and only if $f_1(p,t)(f_2(p,x,t)) = 0$.

where $p_t(x) = 0$ means that executing p on x for t steps ends up in a returning configuration (s,h,κ) with $s(r_1) = 0$. The output of f_2 is called the witness trace.

The first condition ensures that there is an efficient way of translating arbitrary instances of *MRSAT* to instances of *ECSAT*, in a way, such that either both are yes-instances of their respective problems, or none of them are. The second condition ensures that there is an efficient way of translating arbitrary program inputs to circuit inputs, such that either both inputs witness the satisfiability of their model of computation (program or circuit), with respect to some time bound t, or none of them do.

4.2 Circuit reduction

In this section I describe how the circuit compiler works. The naive approach for transforming programs into arithmetic circuits is to directly translate the control-flow graph (CFG) of the program into a circuit, which is the approach taken by [HK20b]. This has the disadvantage, that since circuits



Program: $\mathbf{Const}_{enc(i_1)}, \mathbf{Const}_{enc(i_2)}, \ldots, \mathbf{Const}_{enc(i_l)}$
S_{init}
S_1
S_2
S_{t-1}
S_t
C_{TR}
Transition circuit
Memory consistency circuit
adr_1 val_1 $ldr?_1$ adr_2 val_2 $ldr?_2$ adr_i val_i $ldr?_i$ adr_t val_t $ldr?_t$
$C_1, C_2, \ldots, C_k$
Const_1
Const_2
Const_t
$C_{Waksman}$
$\sigma(1)$ $\sigma(2)$ $\sigma(i)$ $\sigma(t)$
$adr_{\sigma(1)}$ $adr_{\sigma(2)}$ $adr_{\sigma(i)}$ $adr_{\sigma(t)}$ $val_{\sigma(1)}$ $val_{\sigma(2)}$ $val_{\sigma(i)}$ $val_{\sigma(t)}$ $ldr?_{\sigma(1)}$ $ldr?_{\sigma(2)}$ $ldr?_{\sigma(i)}$ $ldr?_{\sigma(t)}$
C_{Mem}

Figure 4.5: High-level overview of the circuit that a program p is translated to.

must be input independent, then data-dependent loops in the CFG must be unrolled (perhaps up to some upper bound on the max number of iterations of any loop), and all conditional branches must be part of the description of the circuit.

Recent papers [CHP+23] [HK20a] [BSCTV14] [BSCG+13] take the approach of translating a program p into a circuit C, where the input to C is a trace tr of the execution of p on some $x \in \text{Word}^n$, and C works by composing two sub-circuits that verifies that

1. that operations of the machine's arithmetic logical unit (ALU) is consistent with tr and p, where *memory operations are ignored*, and

2. that the values read/written to memory in the trace tr are sequentially consistent (every read from a location l sees the most recent write to l).

The first sub-circuit is sometimes called the *transition circuit*, the second sub-circuit the *memory consistency circuit*, and tr is often called the *witness trace*. The theoretical foundations of this paradigm were to the best of my knowledge contributed in [BSCGT12], and subsequently optimized and implemented for the first time in [BSCG+13].

Figure 4.5 gives a high-level overview of the output of the circuit compiler, when run on a program p and time bound t: The program p is first encoded as constants as explained in section 4.2.1. Then t copies of the circuit C_{TR} are wired to make up the transition circuit, which takes as input the *local states* $S_1, \ldots, S_t$ that results from encoding a witness $x \in \text{Word}^n$. Each circuit C_{TR}

outputs a memory address, MiniRAM word (a value), and a boolean signifying if the instruction at that step performed a read from memory. That is

- $adr = l+1$, $val = w$ if the instruction corresponding to the current local state was a read from (**ldr**) or write to (**str**) memory with value w at address l. In case the instruction read from memory then $ldr? = 1$.
- $adr = val = ldr? = 0$ otherwise.

The reason that local states which reads from/ writes to memory address l outputs $adr = l+1$ is to distinguish reads from/ writes to address 0 from instructions that don't access memory. This is the reason why the operational semantics of MiniRAM imposes the constraint that valid memory addresses l must be no higher than $2^{32}-2$, since the circuit is over $\mathbb{Z}_{2^{32}}$ and we therefore otherwise cannot distinguish accesses to address $2^{32}-1$ from instructions that don't touch memory.

The output of all C_{TR} circuits is the output of the transition circuit, which is the input of the memory consistency circuit. The memory consistency circuit is made up of 4 *permutation networks*, where I'm using arbitrary sized Waksman networks [BD99] as done in [BSCTV14]. In the original TinyRAM paper they mention optimizing the memory consistency circuit by "packing" the memory address, value and step index in a single field element, however I'm unable to do this since my circuits are over $\mathbb{Z}_{2^{32}}$. Construction of the networks as well as a simple routing algorithm based on bipartite graph coloring is explained in section 4.2.3. Finally, the circuits C_{Mem} for checking sequential consistency of memory accesses is explained in section 4.2.3.

4.2.1 Instruction encoding and decoding

Each instruction is encoded as 5 fields spanning 2 words, as illustrated in figure 4.6. The first word consists of the following 4 fields:

Field #1	Contains the instruction's *opcode*.
Field #2	Contains the instruction's destination register.
Field #3	Contains the instruction's first operand, if any, which is always a register.
Field #4	Contains unused bits, that is, padding whose sole purpose is to fill the word.

The second word contains a single operand, which is either a register or word. To simplify the instruction decoding circuit, then I'm using two opcodes for instructions that operate on both registers and words, so 10 opcodes are needed using $\lceil \log 10 \rceil = 4$ bits of the first field of the instruction. Since the machine has 16 registers then 4 bits are needed for field #2 and #3, but for ease of decoding (and to add more registers to the machine in the future without changing the instruction serialization format), then I'm using 8 bits for each of the first 4 fields.

The opcodes for each instruction is given in figure 4.7. There are a few things to notice about them - I have chosen binary representations that eases the memory-consistency checks, in particular, they have the following three properties:

1. The least-significant-bit (lsb) is 1 only if the instruction reads from data-memory.

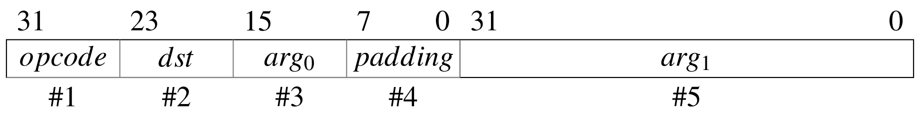


Figure 4.6: Instructions are encoded as two 32-bit words containing 5 fields.

Instruction	arg_1 kind	Opcode
str		$000010_2 = 2_{10}$
ldr		$000011_2 = 3_{10}$
add		$000100_2 = 4_{10}$
sub		$001000_2 = 8_{10}$
mov	register	$001100_2 = 12_{10}$
mov	constant	$010000_2 = 16_{10}$
b		$010100_2 = 20_{10}$
b Z		$011000_2 = 24_{10}$
ret	register	$100000_2 = 32_{10}$
ret	constant	$100100_2 = 36_{10}$

Instruction	field #2	field #3	field #5
str r_{dst} r_{src}	r_{src}	r_{dst}	0
ldr r_{dst} r_{src}	r_{dst}	r_{src}	0
add r_{dst} r_{lhs} r_{rhs}	r_{dst}	r_{lhs}	r_{rhs}
sub r_{dst} r_{lhs} r_{rhs}	r_{dst}	r_{lhs}	r_{rhs}
mov r_{dst} r_{src}	r_{dst}	0	r_{src}
mov r_{dst} w	r_{dst}	0	w
b r	pc	0	r
b Z r	pc	0	r
ret r	r_1	0	r
ret w	r_1	0	w

Figure 4.7: *Left*: Encoding of the opcode. The lsb is 1 only if the instruction is a **ldr**, which is convenient for designing the memory consistency circuit. Similary, the bit in position 1 (0-indexed, from the right) is 1 only if the instruction operates on memory. The msb is set, only if the instruction is a **ret**. *Right*: Encoding of the operands of each instruction into fields #2, #3 and #5 of the serialized instruction. Field #2 encodes the register that is updated by the instruction, with the exception of **str** instructions. Field #3 contains the address from memory read from/ written to by **str** and **ldr**.

2. The bit in position 1 is 1 only if the instruction reads from/ writes to data-memory.
3. The most-significant-bit (msb) is 1 only if the instruction is a **ret**.

Encoding of operands and instruction decoding Figure 4.7 gives the encoding of instruction's operands in fields #2, #3 and #5. The encoding follows the following two principles:

- Field #2 contains the register that is *being updated* by the instruction, with one exception being **str** instructions, since **str** only updates the contents of memory and not any registers.
- Field #3 contains the address read from/ written to for **str** and **ldr**.

Figure 4.8 gives the circuits for decoding the two words of the serialized instruction. Its construction is straight forward based on the instruction format: The left circuit takes as input the first (low) word of the instruction $instr_{low}$ and decodes the contents of field #1, #2 and #3 using Encode4 gates. Based on the encoding of the opcodes in figure 4.7 then $ldr?$ is 1 only if the instruction is a **ldr**, $mem?$ is 1 only if the instruction is a **ldr** or **str** and $ret?$ is 1 only if the instruction is a **ret**. The right circuit takes as input the second (high) word of the instruction $instr_{hi}$

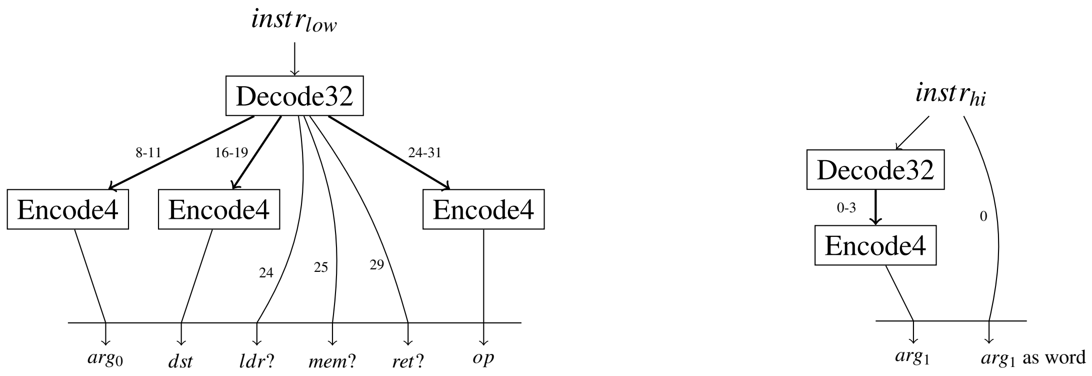


Figure 4.8: Circuit C_{Decode} for instruction decoding is composed of two sub-circuits. The left circuit decodes the first word of an instruction, according to the instruction format and opcode encoding (as given in figure 4.7). The right circuit decodes the second word of an instruction, where the first output is used (in the ALU sub-circuit) if the operand to the instruction is a register and the second output is used if the operand is a constant word.

and encodes uses an Encode4 gate to truncate the value to a "well defined" register, which is used if the instruction operates on registers, as well as outputting the input word, which is used if the instruction operates on words.

4.2.2 Transition circuit

Figure 4.9 gives the sub-circuit C_{TR} which is composed t times in the final transition circuit (see figure 4.5). The circuit takes as input two consecutive local states from the witness trace, consisting of the values of all registers and conditional flags before and after executing the i'th instruction. At a high level, then C_{TR} performs operations similar to a single iteration of the fetch-decode-execute loop in a CPU, while also verifying that that the resulting state matches the expected state S_{i+1}:

- **Fetch**: The circuit C_{Fetch} fetches the high and low words of the encoded instruction, by selecting corresponding hard-coded constant words using the program counter from S_i.
- **Decode**: The circuit C_{Decode} (figure 4.8) decodes the instruction. The value of all encoded registers is is fetched in $C_{LoadFromReg}$, by selecting the corresponding indices from the local states. To verify correctness of the execution step, then the value of the destination register from S_{i+1} is also loaded. Finally, C_{Decode} also returns the previously described values $ldr?$, $mem?$, $ret?$ which is used to verify consistency of all registers ($C_{CheckRegs}$) and to compute the address and value loaded for memory operations ($C_{MemAdrVal}$).
- **Execute**: The circuit C_{ALU} receives as input the opcode op, the 32-bit value of field #5 of the instruction arg_{1Word} as well as value of registers arg_0, arg_1 and dst from S_i and dst from S_{i+1}. The circuit then computes the expected value for each possible instruction, mocking the result of memory operations using the expected result of dst from S_{i+1}. The circuit then outputs the expected result by selecting based on the opcode op. The expected value of the conditional

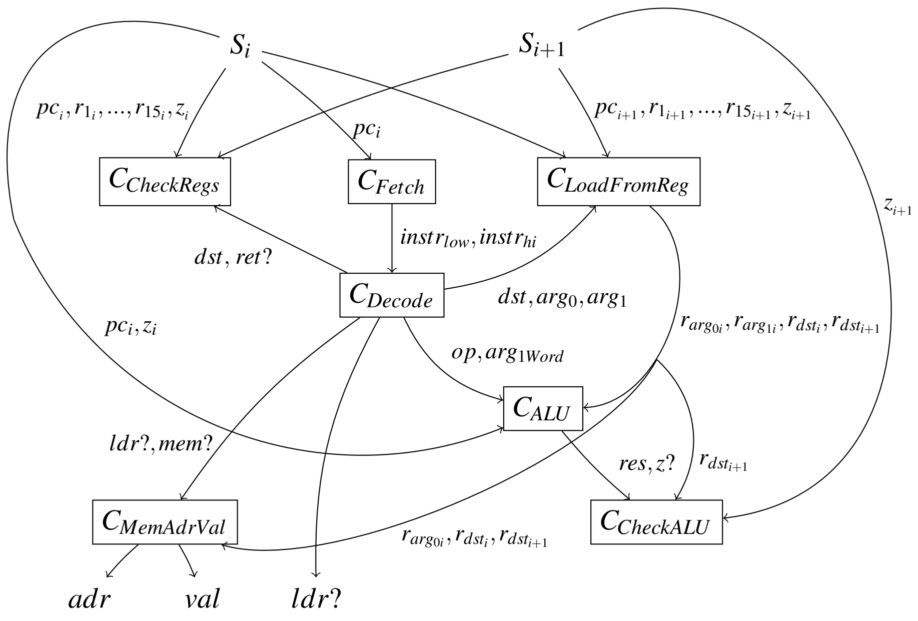


Figure 4.9: The circuit C_{TR}

zero flag $z?$ by computing the "arithmetic or"[4] of all bits in the result and computing the negation this.

- **Verify**: The following circuits verify that executing the decoded instruction with local state S_i results in the state S_{i+1}:

  1. $C_{CheckRegs}$ ensures that all registers are consistent, meaning

$$\begin{aligned} r_{k_i} &= r_{k_{i+1}} && \text{if } k \neq dst \quad \text{and} \\ pc_{k_{i+1}} &= \begin{cases} pc_{k_i} + 1 & \text{if } ret? = 0 \\ pc_{k_{i+1}} = pc_{k_i} & \text{otherwise.} \end{cases} \end{aligned}$$

  2. $C_{CheckALU}$ ensures that the result of C_{ALU} is correct, meaning $res = r_{dst_{i+1}}$ and $z? = z_{i+1}$.

All sub-circuits are given in appendix B.

4.2.3 Memory consistency circuit

The memory consistency circuit ensures that memory operations are sequentially consistent, meaning each read from a location l sees the most recent write to l[5].

The naive way of designing this, is to maintain a snapshot of the entire memory of the machine, leading to circuits of size $\mathcal{O}(t^2)$. This is the approach taken by the Fairplay compiler [MNPS04], however [BSCG$^+$13] made the observation, that if the execution trace is sorted on memory accesses with ties broken by timestamp, then it's much easier to verify that memory operations are sequentially consistent, and a linear scan suffices: If l is a location that's accessed in time t_i and t_j with $t_i < t_j$ and the access at t_j loaded a value v, then v must be equal to the value loaded or stored at time t_i.

A permutation network is used to sort the execution trace, and the prover commits to the configuration of the network, which computes the sorting permutation according the order from above, as input to the circuit.

C_{Mem} from appendix B shows the final memory consistency circuit. The right-most output (which must be 0) verifies memory accesses, and the rightmost output verifies that the inputs are sorted. In the next section I will explain how arbitrary sized Waksman networks are constructed, and I'll give the algorithm used to compute a configuration of the network, so that it computes a given (sorting) permutation.

Permutation network

An arbitrary sized (AS) Waksman network [BD99] is a network with n inputs and outputs, that computes a *rearrangable* permutation of the inputs. The network consists of "switches", each of which takes 2 inputs, produces 2 outputs and can be in 1 of 2 positions: In position $b \in \{0,1\}$

[4]Using "arithmetic xor" of two bits $f(x,y) = x + y - 2xy$ and De Morgans law.

[5]As previously mentioned, if l is uninitialized, then a read from l may return any value, similar to the semantic of uninitialized reads in a low-level language like C.

the switch sends input b to output $1-b$. The switches are connected in a way, so that for any permutation $\sigma \in S_n$ there exists a configuration of the switches, so that the network maps an input $x \in U^n$ to $\sigma(x)$.

Here, U is some set of elements the network operates on. For example, the original motivation behind these networks took U to be the (infinite) set of audio or video signals, and the purpose of the network was to solve some problem in the design of on-board networks in telecommunication satellites, where signals needed to be routed to specific (but dynamically changing) amplifiers. In our case then $U = \mathbb{Z}_{2^{32}}$, and the network will be used to route memory addresses along with the value read/ written at that address.

Construction An AS-Waksman network of size n is constructed recursively by two networks of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. This is illustrated for $n=3$ and $n=4$ in figure 4.11, and the general recursive construction is given in figure 4.12. This leads to a straight-forward recursive algorithm, whose running time is proportional to the number of switches in the circuit. Let's analyze this number:

If we let $P(n)$ denote the number of switches needed to construct a network of size n, then $P(1)=0$, since the network (called a *link*) just connects its input and output. From figure 4.12 it's easy to count that $n-1$ new switches are added when constructing a network of size n. Furthermore, since the construction is recursively connecting sub-networks of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, then we obtain the recurrence $P(n) = P(\lceil n/2 \rceil) + P(\lfloor n/2 \rfloor) + n - 1$. From this it's not too hard to show by induction [BD99] that $P(n) = \sum_{i=1}^{n} \lceil \log i \rceil$ and that $P(n)$ is bounded from above by $n \log n - 0.91n + 1$ for $n \geq 1$. Thus, the construction is asymptotically optimal, since a simple counting argument shows that the number of switches necessary to construct a rearrangable network of size n, that can realize all $n!$ permutations in S_n, is bounded from below by $\lceil \log n! \rceil$[6], and $\log n! = n \log n - n \log e + \mathcal{O}(\log n)$ by Sterling's formula.

Routing When constructing the witness trace, one needs to solve the following problem: Given an AS-Waksman network of size n, and a permutation $\sigma \in S_n$, compute a configuration for the switches of the network, so it routes an input $x \in \mathbb{Z}_{2^{32}}^n$ to $\sigma(x)$.

By observing that input $2i-1$ and $2i$ of the network must be send to different sub-networks, for $i = 1, .., \lfloor n/2 \rfloor$, and, similarly, output $2j-1$ and $2j$ must come from different sub-networks, for $j = 1, ..., \lfloor n/2 \rfloor$, then one can reduce this to the problem of 2-coloring the vertices of a bipartite graph[7]: Let $p \in \mathbb{N}$ so that $n = 2p$ or $n = 2p+1$. Construct the graph $G_\sigma = (V, E)$ by defining $V = \{1, 2, ..., 2p\}$ and E as the set containing an edge for each switch of the input layer

$$(2i-1, 2i) \in E \quad \text{for} \quad i = 1, 2, ..., p$$

as well as an edge for each switch of the output layer

[6]it must be possible to configure the network in $n!$ distinct ways, and each switch can be configured in 2 ways

[7]A graph (V, E) is bipartite if V can be divided into disjoint sets A and B, so every edge connects a vertex in A with one in B. The problem of 2-coloring the vertices of a graph is to assign a color $c \in \{0, 1\}$ to each vertex, so u and v have different colors if they are connected by an edge. Every bipartite graph can be 2-colored, which is not so hard to see: Assign the color 1 to each vertex in A and the color 2 to each vertex in B. This is a valid coloring since the graph is bipartite.

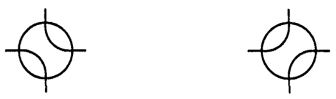


Figure 4.10: Possible connections (states) of a physical rotative switch from [BD99]. In our case these are of course implemented in software, by computing the function $f_b(x,y) = (x(1-b)+yb, xb+y(1-b))$ for a constant configuration parameter $b \in \{0,1\}$.

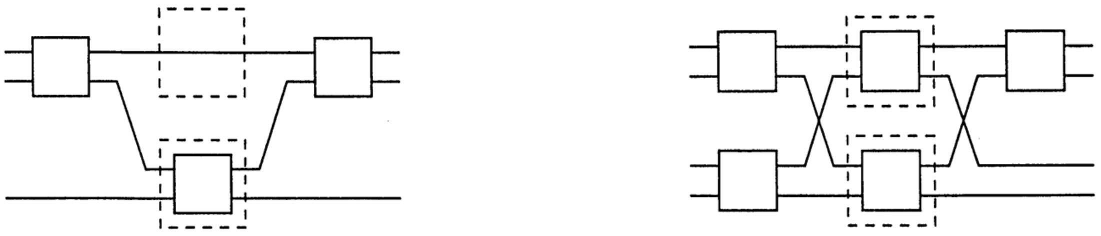


Figure 4.11: Left: AS-Waksman network of size 3. The network can be seen as being build from two networks of size 2 (a switch) and 1 (a link) respectively, as indicated by the dashed boxes. Right: AS-Waksman network of size 4, which can be seen as being build from two networks of size 2 (two switches). Both illustrations are from [BD99].

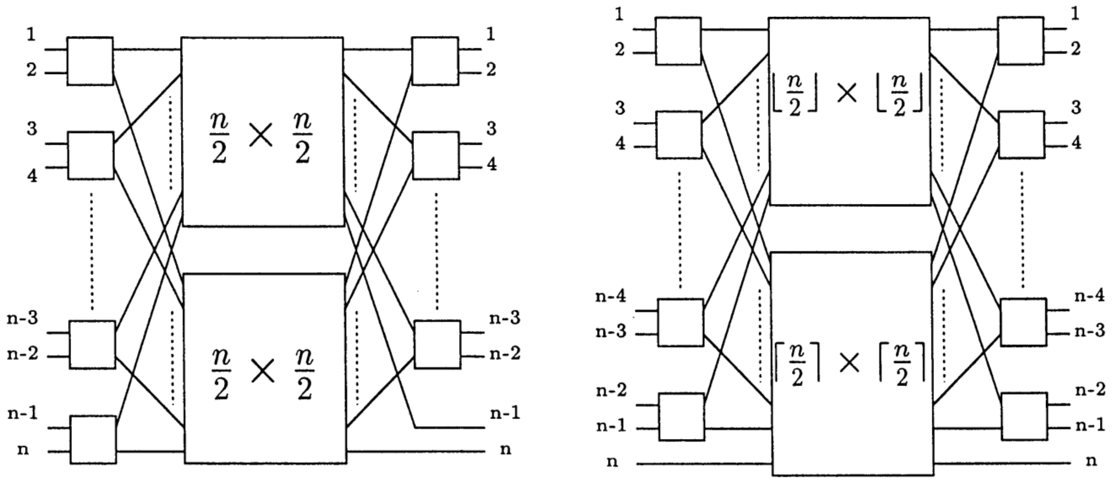

1
2
3
4
n-3
n-2
n-1
n
$\frac{n}{2} \times \frac{n}{2}$
$\frac{n}{2} \times \frac{n}{2}$
1
2
3
4
n-3
n-2
n-1
n
1
2
3
4
n-4
n-3
n-2
n-1
n
$\lfloor \frac{n}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$
$\lceil \frac{n}{2} \rceil \times \lceil \frac{n}{2} \rceil$
1
2
3
4
n-4
n-3
n-2
n-1
n

Figure 4.12: General construction of AS-Waksman networks. The left construction is when the size n of the network is even, and the right is for odd values of n. In both cases $n-1$ new switches are used and two sub-networks of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ needs to be constructed, leading to the recurrence $P(1) = 0$ and $P(n) = P(\lceil n/2 \rceil) + P(\lfloor n/2 \rfloor) + n - 1$ where $P(n)$ is the number of switches needed to construct a network of size n. One can show that $P(n) = \sum_{i=1}^{n} \lceil \log i \rceil$.

$$(\sigma(2i-1), \sigma(2i)) \in E \quad \text{for} \quad i = 1, 2, ..., p$$

where, in case n is even, the edge $(\sigma(2p-1), \sigma(2p)) \in E$ corresponds to a switch with a fixed configuration. When coloring the graph G_σ then $\sigma(2p-1)$ is always assigned the color 0 and $\sigma(2p)$ the color 1, so last two outputs are "hard-coded" to come from the upper and lower network respectively. This is not a restriction when n is odd, and a coloring could have $\sigma(2p)$ be outputted from the lower network, since the color of each vertex can then just be flipped.

From this definition, then G_σ has a max degree[8] of 2, in fact, if n is even then all vertices have a degree of 2: The edges for the input layer contribute 1 to the degree of all vertices, and since σ is a permutation, then the edges of the output layer also contribute 1 to the degree of all vertices of the graph.

One can show that the graph only has even-length cycles, so it is bipartite and can be 2-colored with the following simple algorithm from [Hus15]. The algorithm has been adapted to work when the input is not necessarily a connected graph, and parameterized with a starting vertex whose color is 1, to distinguish between the case when n is odd or even:

Algorithm: **Bipartition**

On input a bipartite graph $G = (V, E)$ and starting vertex $u \in V$ this algorithm finds a 2-coloring represented as a mapping $f : V \to \{0, 1\}$:

1. **Initialize** Let $f(u) = 1$ and let Q be an empty queue. For each neighbor w of u set $p(w) = u$ (the 'parent' of w) and add w to Q.

2. **Next vertex** If Q is empty, go to step 3. Otherwise remove the first vertex v from Q and set $f(v) = 1 - f(p(v))$ (the color not assigned to v's parent). For each neighbor w of v if $f(w)$ is undefined (meaning w is not colored yet) and $w \notin Q$ then set $p(w) = v$ and add w to the end of Q. Repeat step 2.

3. **Next connected component** If $f(v)$ is defined for all $v \in V$, return f. Otherwise add some uncolored vertex v to Q and repeat step 2.

Return the 2-coloring f.

The algorithm performs a basic graph traversal and runs in time $\mathcal{O}(|V| + |E|)$. Therefore computing the configuration of the input and output layer of a AS-Waksman network of size n can be done in time $\mathcal{O}(n)$, since $|V| = n$ and $|E| = 2p$ where $n = 2p$ or $n = 2p + 1$.

The algorithm *Route* computes the configuration of an entire AS-Waksman network of size n (not just the input and output layers), using *Bipartition* as a subroutine:

[8]The max degree of a graph is the max number of edges incident to any single vertex.

Algorithm: **Route**

On input two sequences x and y of length n, where x is a permutation of y and where $n = 2p$ or $n = 2p+1$:

1. Construct the bipartite graph $G_\sigma = (V,E)$ where $V = \{x_1, x_2, ..., x_n\}$ and
$$(x_{2i-1}, x_{2i}), (y_{2i-1}, y_{2i}) \in E \quad \text{for } i = 1,2,...,p.$$
2. Run $Bipartite(G_\sigma, y_{2p})$ to compute a 2-coloring $f : V \to \{0,1\}$ of G_σ where $f(y_{2p}) = 1$. Define configuration bits for the input and output layer as $c_i = f(i)$ for $i = 1,2,...,2p$.
3. Construct sequences x', y' of length $2p$ by letting
$$x'_i = \begin{cases} x_{2i-1} & \text{if } f(x_{2i-1}) = 0 \\ x_{2i} & \text{otherwise} \end{cases} \quad , \quad x'_{p+i} = \begin{cases} x_{2i-1} & \text{if } f(x_{2i-1}) = 1 \\ x_{2i} & \text{otherwise} \end{cases}$$
and
$$y'_i = \begin{cases} y_{2i-1} & \text{if } f(y_{2i-1}) = 0 \\ x_{2i} & \text{otherwise} \end{cases} \quad , \quad y'_{p+i} = \begin{cases} y_{2i-1} & \text{if } f(y_{2i-1}) = 1 \\ y_{2i} & \text{otherwise} \end{cases}$$
for $i = 1,2,...,p$.
4. Recursively route $x'_1, x'_2, ..., x'_p$ and $y'_1, y'_2, ..., y'_p$ to receive configuration bits $c_{2p}, c_{2p+1} ..., c_a$ where $a = 2p + \sum_{i=1}^{p} \lceil \log i \rceil$.
5. Recursively route $x'_{p+1}, x'_1, ..., x'_{2p}$ and $y'_{p+1}, y'_1, ..., y'_{2p}$ to receive configuration bits $c_{a+1}, c_{a+2}, ..., c_{2(a-p)}$.

Return the configuration $c_1, c_2, ..., c_{2(a-p)}$.

Given some permutation $\sigma \in S_n$ then on input $1,2,...,n$ as well as $\sigma(1), \sigma(2), ..., \sigma(n)$ to *Route*, the algorithm returns a configuration $c_1, c_2, ..., c_{P(n)}$ for the AS-Waksman network of size n, so that the network routes input's according to σ. The size of the configuration is obviously the same as the switch count $P(n)$. Since *Bipartition* runs in linear time, then the running time of the algorithm is proportional to the switch count $P(n)$, which as mentioned is bounded from above by $n \log n - 0.91n + 1$. Thus the asymptotic running time of *Route* is $\mathcal{O}(n \log n)$.

4.2.4 Constructing the witness trace

On input a program p, some $x \in \text{Word}^n$ and time bound t, the function f_2 (from section 4.1.1) interprets x for t steps according to the transition relation, and records the following values for each step:

1. The value of all registers $pc, r_1, ..., r_{15}$.
2. The value of all conditional flags (currently just the zero flag Z).
3. For **ldr** and **str** instructions: The address l and value read/written at l.

I will refer to the value of registers and conditional flag as the *local state* at step i. Name these states $S_1, S_2, ..., S_t$. The function then sorts $S_1, S_2, ..., S_t$ according to the values recorded for the memory accesses, and with the following ordering:

1. $S_i < S_j$ if S_i touched memory and S_j didn't.
2. Let l_i and l_j be the address of memory operations for local states S_i and S_j respectively:
   (a) If $l_i < l_j$ then $S_i < S_j$.
   (b) If $l_i = l_j$ then $S_i < S_j$ are ordered according to their indices: If $i < j$ then $S_i < S_j$.

That is, the local states are sorted first on memory address accessed, and then on step index (or "timestamp"). Call the resulting indices $i_1, i_2, ..., i_t$ so that $S_{i_j} < S_{i_{j+1}}$ for $j = 1, .., t-1$.

The function now runs the subroutine *Route* with input sequences $1, 2, ..., t$ and $i_1, i_2, ..., i_t$ to compute the configuration $c_1, c_2, ..., c_k$ of the AS-Waksman permutation network of size t that sorts $S_1, S_2, ..., S_t$ according to the above ordering. So $k = \sum_{i=1}^{t} \lceil \log i \rceil$.

Finally, the function returns the witness trace as the concatenation of the local states $\mathcal{S} = S_1, S_2, ..., S_t$ and AS-Waksman network configuration $\mathcal{C} = c_1, c_2, ..., c_k$.

Chapter 5

Evaluation

I've implemented an interpreter for MiniRAM and ZKPoK for *MRSAT* in Rust, the sources can be found at `https://www.github.com/mzacho/miniram`. The implementation has been mighty helpful in understanding VOLE-based ZK proof systems and reality-checking my understanding of the theory. In order to evaluate the system, I'm asking the following research questions, which addresses the practical performance and usability of the system:

Q1 How well does the approach scale? What are the bottlenecks?

Q2 Is the compiler useful for solving real-world cryptographic problems?

Performance To address Q1 I've first analyzed the quality of the circuit compiler, by measuring the size of the circuit, so the total number of (non-linear) gates as a function of the time bound t. This identifies, not surprisingly, the sub-circuit $C_{Waksman}$ as the bottleneck when verifying programs for many steps.

Figure 5.1 shows the size of the sub-circuits C_{TR}, C_{Mem} and $C_{Waksman}$ that results from the circuit compiler. As figure 4.5 shows, then the size of C_{TR} and C_{Mem} don't depend on the time bound t, but their contribution to the size of the transition circuit (resp. memory circuit) is linear in t. As t increases then the 4 replicated $C_{Waksman}$ circuits dominate the size of the final compiled circuit, as the number of non-linear gates are linear in the switch count $P(t)$ of the network, which is $\mathcal{O}(t \log t)$.

Usability To address Q2 I've investigated how convenient MiniRAM is for implementing SHA256[1]. Since the algorithm uses operations $SHR_n(x)$ and $ROTR_n(x)$ for respectively shifting and rotating $x \in \mathbb{Z}_{2^{32}}$ right by a constant $n \in \mathbb{Z}_{32}$, as well as bitwise XOR and AND, then I've extended the instruction set of MiniRAM to include corresponding instructions. The circuit compiler has then been adapted, so the circuit C_{ALU} verifies the new instructions in the obvious way, using **Decode32** gates and arithmetic XOR sub-circuits.

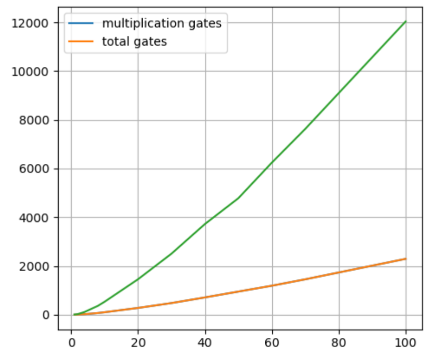
The hash function was implemented without any issues[2]. The 32-bit word size in SHA256 and the wrapping semantics of addition and multiplication conveniently matches the architecture and

[1] As described in FIPS 180-4.

[2] besides my mistake of following an old version (FIPS 180-2) of the spec where bitwise complement has been encoded as U+0020 SPACE, which made me bang my head against the wall for few days.

Circuit size	C_{TR}	C_{Mem}
n input gates	17	(depends on t)
a multiplication gates	165	701
b select alternatives	201	0
c select const alternatives.	$4l$	0
d decode32 gates	5	4
e CheckAllEqButOne pairs	16	0
f output gates	2	2
total number of gates	$3825+2l$	5565
constants	$3+2l$	0

(a) The size of the circuits C_{TR} and C_{Mem}. l is the number of instructions and t is the time bound.



(b) The size of the circuit $C_{Waksman}$ as the time bound t increases. The only non-linear gates in $C_{Waksman}$ are multiplication gates.

Figure 5.1: The size of the sub-circuits that the circuit compiler outputs (see figure 4.5).

		Execution time (ms)		Circuit size					
t	VOLE size	$\mathcal{P}$	$\mathcal{V}$	**Mul**	**Decode32**	**Select** alts.	**SelectConst** alts.	**CheckAllEqButOne** pairs	total gates
100	170.500	208	59	95.066	896	20.052	495	1.600	981.688
200	345.324	417	114	194.018	1.796	40.152	995	3.200	1.985.736
300	521.372	609	171	294.122	2.696	60.252	1.495	4.800	2.995.832
400	698.372	817	233	395.122	3.596	80.352	1.995	6.400	4.010.632
500	875.372	1.037	284	496.122	4.496	100.452	2.495	8.000	5.025.432
750	1.321.918	1.571	427	752.430	6.746	150.702	3.745	12.000	7.582.424
1.000	1.768.668	2.093	566	1.008.930	8.996	200.952	4.995	16.000	10.140.424
1.500	2.670.260	3.127	856	1.529.546	13.496	301.452	7.495	24.000	15.296.408
3.000	5.392.444	6.434	1.795	3.107.778	26.996	602.952	14.995	48.000	30.850.376
4.500	8.130.812	9.484	2.770	4.701.242	40.496	904.452	22.495	72.000	46.484.312
6.000	10.887.812	13.590	3.578	6.312.242	53.996	1.205.952	29.995	96.000	62.210.312
8.000	14.563.812	18.352	4.667	8.460.242	71.996	1.607.952	39.995	128.000	83.178.312
10.000	18.270.548	22.613	5.745	10.637.170	89.996	2.009.952	49.995	160.000	104.298.184
15.000	27.545.548	36.605	9.848	16.087.170	134.996	3.014.952	74.995	240.000	157.138.184
30.000	55.560.202	70.945	19.201	32.655.026	269.996	6.029.952	149.995	480.000	316.801.928
t	superlinear ($\mathcal{O}(t \log t)$)				$9t-4$	$201t-48$	$5t-5$	$16t$	superlinear

Figure 5.2: Prover and verifier execution times for increasing time bounds when proving program with a fixed (1) number of instructions.

Circuit size	1 iteration: $\lvert x\rvert < 8 \cdot 56, t = 3889$	2 iterations: $\lvert x\rvert < 8 \cdot 120, t = 7646$
n input gates	111.180	221.189
a multiplication gates	4.139.328	8.080.046
b select alternatives	798.927	1.536.798
c select const alternatives.	63.222.366	121.624.305
d decode32 gates	35.771	68.810
e CheckAllEqButOne pairs	63.600	122.336
f output gates	15.898	30.582
total number of gates	104.192.872	201.053.056
constants	7.957	7.957
Pre-processing		
VOLE size	70.279.001 (1.1Gb keys/tags)	135.323.393 (2.1Gb keys/tags)
Witness encoding time	20.172ms	75.707ms
Circuit compilation time	1.379ms	2.953ms
Online phase		
$\mathcal{P}$ time (total)	48.273ms	98.154ms
$\mathcal{P}$ time (final mult. check)	2.804ms	5.473ms
$\mathcal{V}$ time (total)	8.671ms	16.784ms
Communication $\mathcal{P}$ to $\mathcal{V}$ ($\approx\theta$)	1.1Gb	2.1Gb
Communication $\mathcal{V}$ to $\mathcal{P}$ ($\approx\theta$')	1.3Gb	2.6Gb

Figure 5.3: The circuit size resulting verifying 1 round of SHA256 hashing. $\theta = a+2b+c+32d+e$ and $\theta' = \theta+b+c+d+2e+f$ as defined in 3.3.3

semantics of MiniRAM. Programming in MiniRAM is also made easier by using a builder pattern and meta-programming the rounds function in Rust, although it would be more convenient to have a compiler take care of register allocation and branching targets.

By modifying the program to return $SHA256(x) - MAC$ for a public constant MAC, my ZKPoK for $MRSAT$ can be used to prove knowledge of a SHA256 pre-image, so some $x \in \{0,1\}^{2^{64}}$ where $SHA256(x) = MAC$, in zero-knowledge. Figure 5.3 shows the circuit size, preprocessing time and online execution time, when proving 1 or 2 iterations of SHA256.

A possible application of this could be an online system for authentication, where the user never sends her password over the network, but instead proves in zero knowledge that "she knows" the password in the pre-image of some hash in the applications user database. However the implementation would need some optimization for this to be practical, as 48 seconds is way too long any user is willing to wait to login.

The cost of communicating δ's for (committing to) the witness and output of non-linear gates from $\mathcal{P}$ to $\mathcal{V}$ is $n+\theta$. The only extra communication from $\mathcal{P}$ to $\mathcal{V}$ is U and V in the final multiplication check, where $\mathcal{V}$ sends θ' random 128-bit challenges to $\mathcal{P}$. In total, both parties communicate $\mathcal{O}(t \log t)$ ring elements to each other.

When increasing t, bounds the number of rounds of hashing used to compute MAC, that the prover reveals to the verifier, the bottleneck on my 16GB laptop is memory usage: Running all three parties simultaneously I run out memory when trying to prove three rounds of hashing ($t = 11.403$). To scale this further I'd need to free up unused memory more aggressively, or stream circuit values to disk, however the latter will slow down the program considerably.

Chapter 6

Conclusion

I've given the syntax and operational semantics of a MiniRAM, a RISC machine designed for efficient and convenient zero-knowledge verification and compilation to arithmetic circuit, similar to the approach of TinyRAM, by emulating the CPU and memory of the machine. The compiler produces arithmetic circuits over $\mathbb{Z}_{2^{32}}$ with non-standard gates, designed to ease the compilation process. I've adapted the state-of-the-art zero-knowledge protocol QuarkSilver to work with these gates, which involved changing blinded openings of wires in the circuit to blinded assertions using a multiplication check. I've proved that this is a sound approach, and given simulators to show that the resulting protocol is still actively secure in the $F^{l,s}_{\text{VOLE}}$-hybrid model.

To demonstrate that the system is useful for giving zero-knowledge proofs for statements of algorithmic nature (as opposed to e.g. proving that a number is a square modulo a prime), I've implemented SHA256 in my Rust implementation of MiniRAM, and used this to prove knowledge of a SHA256 pre-image that's at most 55 bytes long (and thus takes 1 round to hash). This took 48 seconds to prove and 9 seconds to verify, excluding VOLE pre-processing time.

Bibliography

[AHIV23] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: lightweight sublinear arguments without a trusted setup. *Des. Codes Cryptography*, 91(11):33793424, jul 2023.

[BBMH$^+$21] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoit Razet, and Peter Scholl. Appenzeller to brie: Efficient zero-knowledge proofs for mixed-mode arithmetic and $\mathbb{Z}_{2^k}$. Cryptology ePrint Archive, Paper 2021/750, 2021. `https://eprint.iacr.org/2021/750`.

[BBMHS22] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, and Peter Scholl. Moz$\mathbb{Z}_{2^k}$arella: Efficient vector-ole and zero-knowledge proofs over $\mathbb{Z}_{2^k}$. Cryptology ePrint Archive, Paper 2022/819, 2022. `https://eprint.iacr.org/2022/819`.

[BCG$^+$19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. Cryptology ePrint Archive, Paper 2019/1159, 2019. `https://eprint.iacr.org/2019/1159`.

[BCGI19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. Cryptology ePrint Archive, Paper 2019/273, 2019. `https://eprint.iacr.org/2019/273`.

[BD99] Bruno Beauquier and Éric Darrot. On arbitrary waksman networks and their vulnerability. 1999.

[BDSW23] Carsten Baum, Samuel Dittmer, Peter Scholl, and Xiao Wang. Sok: Vector ole-based zero-knowledge protocols. Cryptology ePrint Archive, Paper 2023/857, 2023. `https://eprint.iacr.org/2023/857`.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, page 420, Berlin, Heidelberg, 1991. Springer-Verlag.

[BG92] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In *Annual International Cryptology Conference*, 1992.

[BGI18] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. Cryptology ePrint Archive, Paper 2018/707, 2018. https://eprint.iacr.org/2018/707.

[BMRS20] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. Cryptology ePrint Archive, Paper 2020/1410, 2020. https://eprint.iacr.org/2020/1410.

[BSCG$^+$13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. Cryptology ePrint Archive, Paper 2013/507, 2013. https://eprint.iacr.org/2013/507.

[BSCGT12] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from rams to delegatable succinct constraint satisfaction problems. Cryptology ePrint Archive, Paper 2012/071, 2012. https://eprint.iacr.org/2012/071.

[BSCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct Non-Interactive zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, San Diego, CA, August 2014. USENIX Association.

[CD96] Ronald Cramer and Ivan Damgaard. Linear zero-knowledge - a note on efficient zero-knowledge proofs and arguments. Cryptology ePrint Archive, Paper 1996/004, 1996. https://eprint.iacr.org/1996/004.

[CDE$^+$18] Ronald Cramer, Ivan Damgaard, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spd$\mathbb{Z}_{2^k}$: Efficient mpc mod 2^k for dishonest majority. Cryptology ePrint Archive, Paper 2018/482, 2018. https://eprint.iacr.org/2018/482.

[CHP$^+$23] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. Cheesecloth: Zero-knowledge proofs of real-world vulnerabilities, 2023.

[CO15] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. Cryptology ePrint Archive, Paper 2015/267, 2015. https://eprint.iacr.org/2015/267.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. 1971.

[Dam23] Ivan Damgaard. On sigma protocols, lecture notes on cryptographic protocol theory at aarhus university. 2023.

[DIO20] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. Cryptology ePrint Archive, Paper 2020/1446, 2020. https://eprint.iacr.org/2020/1446.

[DNY23] Ivan Damgaard, Jesper Buus Nielsen, and Sophia Yakoubov. Zero knowledge proofs and arguments. 2023.

[DP97] Damgaard and Pfitzmann. Sequential iteration of interactive arguments and an efficient zero-knowledge argument for np. (50), Jun 1997.

[GMR85] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. 1985.

[Gol04] Oded Goldreich. Foundations of cryptography. ii: Basic applications. 2, 05 2004.

[HK20a] David Heath and Vladimir Kolesnikov. A 2.1 khz zero-knowledge processor with bubbleram. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 20552074, New York, NY, USA, 2020. Association for Computing Machinery.

[HK20b] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. Cryptology ePrint Archive, Paper 2020/136, 2020. `https://eprint.iacr.org/2020/136`.

[Hus15] Thore Husfeldt. Graph colouring algorithms. *CoRR*, abs/1505.05825, 2015.

[JKO13] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: How to prove non-algebraic statements efficiently. Cryptology ePrint Archive, Paper 2013/073, 2013. `https://eprint.iacr.org/2013/073`.

[Kar10] Richard M. Karp. *Reducibility Among Combinatorial Problems*, pages 219–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[Lau03] Niels Lauritzen. *Concrete Abstract Algebra: From Numbers to Gröbner Bases*. Cambridge University Press, 2003.

[Mar03] J.C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill higher education. McGraw-Hill, 2003.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updateable structured reference strings. Cryptology ePrint Archive, Paper 2019/099, 2019. `https://eprint.iacr.org/2019/099`.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—A secure Two-Party computation system. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.

[OSB23] Claudio Orlandi, Peter Scholl, and Carsten Baum. Lecture notes on cryptographic computing at aarhus university. 2023.

[WYKW20] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. Cryptology ePrint Archive, Paper 2020/925, 2020. `https://eprint.iacr.org/2020/925`.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Shaun Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

Appendix A

Proof of theorem 3.3.4

Let C be a circuit over $\mathbb{Z}_{2^k}$ with n **In** gates, a **Mul** gates, b' **Select** gates with a total of b alternatives, c' **SelectConst** gates with a total of c alternatives, d **Decode32** gates, e' **CheckAllEqButOne** gates with a total of e pairs, and f **Out** gates. Let $\theta = a + 2b + c + 32d + e$.

The proof is similar to [BBMHS22], but where the definition of the simulators are fitted to the non-standard gates in the extended arithmetic circuit C.

Proof when $\mathcal{P}$ is corrupted : Define an interactive Turing machine $\mathcal{S}$ with the following behavior:

1. $\mathcal{S}$ simulates the pre-processing phase by choosing $\Delta \in_R \mathbb{Z}_{2^{48}}$ and recording all $n+\theta+1$ values and mac tags from $\mathbb{Z}_{2^{128}}$ that $F_{\text{VOLE}}^{128,48}$ receives from $\mathcal{A}$. $\mathcal{S}$ computes the corresponding keys in the natural way.

2. $\mathcal{S}$ begins to simulate the online phase by receiving $\delta_{r_i} \in \mathbb{Z}_{2^{128}}$ from $\mathcal{A}$ for each input gate $\mathbf{In}_i$ in the circuit, and computing the corresponding value from the input's congruence class $\tilde{w_i} = \delta_{r_i} + \tilde{r}_i \in \mathbb{Z}_{2^{128}}$ as well as $[w_i] = [r_i] + \delta_{r_i}$.

3. $\mathcal{S}$ continues the online simulation gate-by-gate as an honest verifier, but also keeps track of $\mathcal{P}$'s wire values:

    - If the gate is linear (**Add**, **Sub**, **Const**, **MulConst**, **Encode4**, **Encode8** or **Encode32**), then $\mathcal{S}$ computes $\mathcal{P}$'s and $\mathcal{V}$'s part the natural way using the values of the input wires.
    - If the gate is a **Mul** with inputs $\tilde{x}, \tilde{y}$ then $\mathcal{S}$ receives δ_{r_i}, where $[r_i]$ is the next authenticated value, and computes $\tilde{z} = \delta_{r_i} + \tilde{r}_i$ as well as $[z] = [r_i] + \delta_{r_i}$. $\mathcal{S}$ then runs **SimMul**$([x],[y],[z])$.
    - If the gate is a **Select** gate with inputs $\tilde{i}, \tilde{x}_0, \ldots, \tilde{x}_{m-1} \in \mathbb{Z}_{2^{128}}$ then $\mathcal{S}$ receives $\delta_{r_j}, \delta_{r_{j+m}}$ from $\mathcal{A}$ for $j \in [m]$. The simulator computes $\tilde{b}_j = \delta_{r_j} + \tilde{r}_j$ and $\tilde{b}_j \tilde{x}_j = \delta_{r_{j+m}} + \tilde{r}_{j+m}$, where $\tilde{r}_0, \ldots, \tilde{r}_{2m-1}$ are the next unused authenticated values, as well as $[b_j] = [r_j] + \delta_{r_j}$, $[b_{j+m}] = [r_{j+m}] + \delta_{r_{j+m}}$. $\mathcal{S}$ then runs **SimMul**$([b_j],[b_j]-1,[0])$, **SimMul**$([b_j],[i-j],[0])$ and **SimMul**$([b_j],[x_j],[b_{j+m}])$ for all j. $\mathcal{S}$ computes $[b]$ as $[\sum_{j=0}^{m-1} b_j]$ and runs **SimAssert**$([b],1)$ and finally computes the value of the output wire as $\sum_{j=0}^{m-1} [b_{j+m}]$.

- If the gate is a **SelectConst** or **CheckAllEqButOne** then $\mathcal{S}$ proceeds essentially like for **Select** gates, adapting the steps to the differences in the protocol.
- If the gate is a **Decode32** gate with input $\tilde{x}$ then $\mathcal{S}$ receives δ_{r_i} from $\mathcal{A}$ for $i \in [32]$ and computes $[r_i] + \delta_{r_i}$ and $1 - [x_i]$, and then runs $\mathbf{SimMul}([x_i], 1 - [x_i], [0])$. $\mathcal{S}$ then finally runs $\mathbf{SimAssert}([x] - \sum_{i=0}^{31} 2^i [x_i], 0)$.
- If the gate is a **Out** with input $\tilde{x}$ then $\mathcal{S}$ runs $\mathbf{SimAssert}([x], 0)$.
- After computing all gates then $\mathcal{S}$ sends $\vec{\chi} \in_R \mathbb{Z}_s^{\theta'}$ to $\mathcal{A}$ and receives (U, V) as response. It computes $W = \sum_{i=1}^{\theta'} \chi_i B_i + B^* \bmod 2^l$ and checks if $W \neq U - V\Delta \in \mathbb{Z}_{2^l}$, in which case $\mathcal{S}$ sends $(\text{PROVE}, C, \perp)$ to F_{ZK}^k on behalf of $\mathcal{P}^*$ and aborts. Otherwise, $\mathcal{S}$ sends $(\text{PROVE}, C, \vec{w})$ to F_{ZK}^k with $\vec{w}_i = \tilde{w}_i \bmod 2^k$ for $i \in [n]$.

4. $\mathbf{SimMul}([x], [y], [z])$ computes the next B_i as $K[x]K[y] + \Delta K[z] \bmod 2^l$.

5. $\mathbf{SimAssert}([x], y)$ $\mathcal{S}$ receives $\delta_{r_i} \in \mathbb{Z}_{2^{128}}$ from $\mathcal{A}$ and computes $\tilde{x}' = \delta_{r_i} + \tilde{r}_i$ and $[x'] = [r_i] + \delta_{r_i}$ where $\tilde{r}_i$ is the next unused authenticated value. The simulator then computes $[x] - 2^{32}[x'], [1]$ and $[y]$ and runs $\mathbf{SimMul}([x] - 2^{32}[x'], [1], [y])$.

Since $\mathcal{S}$ behaves like an honest verifier towards $\mathcal{A}$, then the trace of $\mathcal{S}$ (the list of messages seen by $\mathcal{A}$ when communicating with $\mathcal{S}$) is indistinguishable from real conversations with the honest verifier in the protocol, and whenever the honest verifier outputs (REJECT) in real conversations with $\mathcal{A}$, then the verifier in the ideal execution also outputs (REJECT), since $\mathcal{S}$ sends (PROVE,C,$\perp$) to F_{ZK}^k. Thus, as in [BBMHS22] we only needs to shown that the probability that the honest verifier accepts the proof in real conversations, but the witness $\vec{w}$ sent to F_{ZK}^k by $\mathcal{S}$ does not satisfy $C(\vec{w}) = 0$. Like [BBMHS22] then we proceed by showing that if $C(\vec{w}) = 0$ then the probability that the honest verifier accepts is at most 2^{-40}.

A corrupt prover can try to cheat in two ways: In the final multiplication checks and in any of the **Assert** sub-protocols. However, theorem 3.3.3 shows that if $\mathcal{A}$ tries to cheat in **Assert** by committing to a value other than $x \gg 32$, then he will have to cheat in the final multiplication checks as well. Thus it suffices to bound the probability that $\mathcal{A}$ cheats in the final multiplication checks:

The argument that $\mathcal{A}$ cannot cheat with probability greater than 2^{-40} is now exactly the same is in [BBMHS22], so we omit it here. The idea is that if the i'th multiplication check contains an error, then the value B_i is the result of evaluating a quadratic polynomial instead of a linear one. Then [BBMHS22] defines a security game identical to this setting, and bounds the probability with which any adversary can win the game, by carefully analyzing the number of solutions to equations of this type.

Proof when $\mathcal{V}$ is corrupted : Define an interactive Turing machine $\mathcal{S}$ with the following behavior:

1. $\mathcal{S}$ simulates the pre-processing phase by recording $\Delta \in_R \mathbb{Z}_{2^{48}}$ and $K[r_i] \in \mathbb{Z}_{2^{128}}$ for $i \in [n + \theta + 1]$ that $F_{\text{VOLE}}^{128,48}$ receives from $\mathcal{A}$.

2. $\mathcal{S}$ begins to simulate the online phase by sending $\delta_{r_i} \in_R \mathbb{Z}_{2^{128}}$ to $\mathcal{A}$ for each input gate $\mathbf{In}_i$ in the circuit and computing $K[w_i] = K[r_i] - \delta_{r_i} \cdot \Delta$.

3. $\mathcal{S}$ continues the online simulation gate-by-gate by sending random values to $\mathcal{A}$ and keeping track of $\mathcal{V}$'s keys:

   - If the gate is linear (**Add**, **Sub**, **Const**, **MulConst**, **Encode4**, **Encode8** or **Encode32**), then $\mathcal{S}$ computes $\mathcal{V}$'s part the natural way using the keys of the input wires.
   - If the gate is a **Mul** and $\mathcal{S}$ holds input keys $K[x]$ and $[y]$, then $\mathcal{S}$ sends $\delta_{r_i} \in_R \mathbb{Z}_{2^{128}}$, and computes $K[z] = K[r_i] - \delta_{r_i} \cdot \Delta$ where $K[r_i]$ is the next unused key. $\mathcal{S}$ then runs **SimMul**$(K[x], K[y], K[z])$.
   - If the gate is a **Select** gate with inputs $K[i], K[x_0], ..., K[x_{m-1}] \in \mathbb{Z}_{2^{128}}$ then $\mathcal{S}$ sends $\delta_{r_j}, \delta_{r_{j+m}} \in_R \mathbb{Z}_{2^{128}}$ to $\mathcal{A}$ for $j \in [m]$. Let $K[r_0], ..., K[r_{2m-0}]$ be the next unused keys. The simulator then computes $K[b_j] = K[r_j] - \delta_{r_j} \cdot \Delta, K[b_{j+m}] = K[r_{j+m}] + \delta_{r_{j+m}} \cdot \Delta$ and $K[i - j] = K[i] - j\Delta$. $\mathcal{S}$ then runs **SimMul**$(K[b_j], K[b_j] + \Delta, 0)$, **SimMul**$(K[b_j], K[i - j], 0)$ and **SimMul**$(K[b_j], K[x_j], K[b_{j+m}])$ for all j. $\mathcal{S}$ computes $K[b] = \sum_{j=0}^{m-1} K[b_j]$ and runs **SimAssert**$(K[b], 1)$ and finally computes the key of the output wire as $\sum_{j=0}^{m-1} K[b_{j+m}]$.
   - If the gate is a **SelectConst** or **CheckAllEqButOne** then $\mathcal{S}$ proceeds essentially like for **Select** gates, adapting the steps to the differences in the protocol.
   - If the gate is a **Decode32** gate with key $K[x]$ then $\mathcal{S}$ sends $\delta_{r_i} \in_R \mathbb{Z}_{2^{128}}$ to $\mathcal{A}$ for $i \in [32]$ and computes $K[r_i] - \delta_{r_i} \cdot \Delta$ and $K[1 - x_i] = -\Delta - [x_i]$, and then runs **SimMul**$(K[x_i], K[1 - x_i], 0)$. $\mathcal{S}$ then runs **SimAssert**$(K[x] - \sum_{i=0}^{31} 2^i K[x_i], 0)$.
   - If the gate is a **Out** with input key $K[x]$ then $\mathcal{S}$ runs **SimAssert**$(K[x], 0)$.
   - After computing all gates then $\mathcal{S}$ receives $\vec{\chi} \in_R \mathbb{Z}_s^{\theta'}$ from $\mathcal{A}$, samples $V \in_R \mathbb{Z}_{2^{128}}$, computes $W = \sum_{i=1}^{\theta'} \chi_i B_i + B^* \bmod 2^l$ and $U = W - V \cdot \Delta$ and sends (U, V) to $\mathcal{A}$.

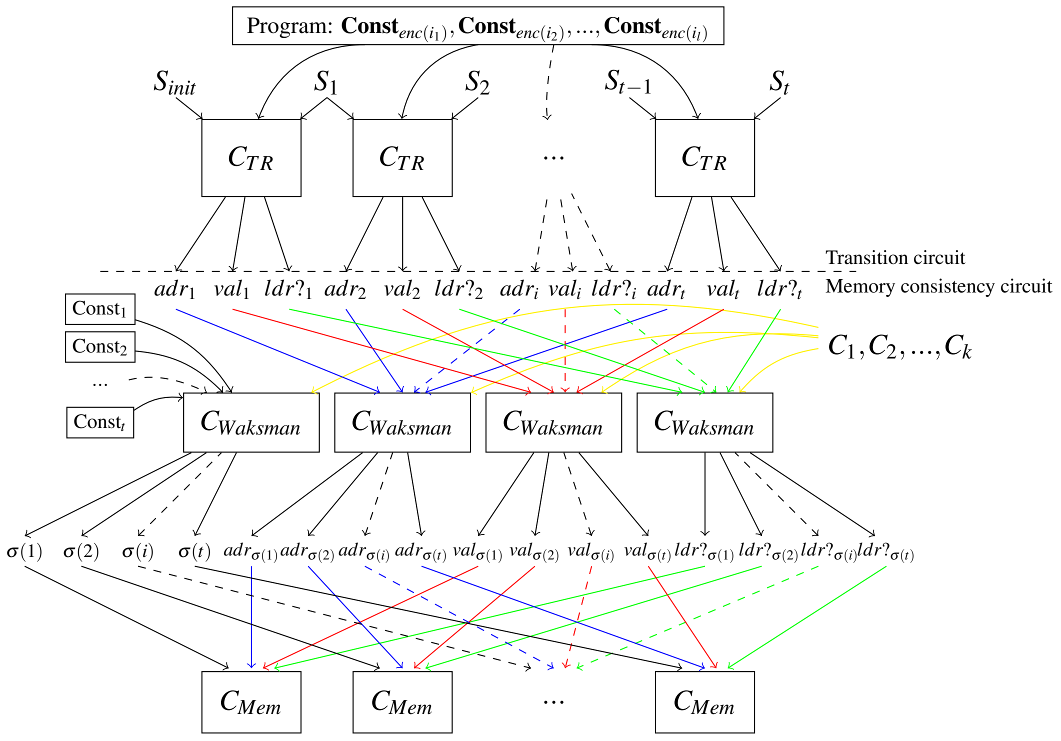
4. **SimMul**$(K[x], K[y], K[z])$ computes the next $B_i = K[x]K[y] + \Delta K[z] \bmod 2^l$ as the honest verifier would do.

5. **SimAssert**$([x], y)$ $\mathcal{S}$ sends $\delta_{r_i} \in_R \mathbb{Z}_{2^{128}}$ to $\mathcal{A}$ and computes $K[x'] = K[r_i] + \delta_{r_i} \cdot \Delta$ where $K[r_i]$ is the next unused key. The simulator then computes $K[1] = -\Delta$ and $K[y] = -y\Delta$ and runs **SimMul**$(K[x] - 2^{32}K[x'], K[1], K[y])$.

The view of $\mathcal{A}$ is distributed exactly as in conversations with the real prover: All δ_{r_i} in the real protocol are random, thus so is $\delta_{x_i} = x_i - \delta_{r_i}$ when committing to x_i in the real protocol. Moreover, so is (U, V) in the real protocol by the same argument (they are masked by random $(A{*}_0, A{*}_1)$), subject to $W = U + V \cdot \Delta$, so (U, V) sent by $\mathcal{S}$ is distributed as in the real protocol, since it keeps track of $\mathcal{A}$'s keys and can compute the value W expected by $\mathcal{A}$. $\square$

Appendix B

Sub-circuits in circuit compiler



Program: $\mathbf{Const}_{enc(i_1)}, \mathbf{Const}_{enc(i_2)}, \ldots, \mathbf{Const}_{enc(i_l)}$
S_{init}
S_1
S_2
S_{t-1}
S_t
C_{TR}
C_{TR}
$\ldots$
C_{TR}
Transition circuit
Memory consistency circuit
adr_1 val_1 $ldr?_1$ adr_2 val_2 $ldr?_2$ adr_i val_i $ldr?_i$ adr_t val_t $ldr?_t$
$C_1, C_2, \ldots, C_k$
Const_1
Const_2
$\ldots$
Const_t
$C_{Waksman}$
$C_{Waksman}$
$C_{Waksman}$
$C_{Waksman}$
$\sigma(1)$ $\sigma(2)$ $\sigma(i)$ $\sigma(t)$ $adr_{\sigma(1)}$ $adr_{\sigma(2)}$ $adr_{\sigma(i)}$ $adr_{\sigma(t)}$ $val_{\sigma(1)}$ $val_{\sigma(2)}$ $val_{\sigma(i)}$ $val_{\sigma(t)}$ $ldr?_{\sigma(1)}$ $ldr?_{\sigma(2)}$ $ldr?_{\sigma(i)}$ $ldr?_{\sigma(t)}$
C_{Mem}
C_{Mem}
$\ldots$
C_{Mem}

Figure B.1: High-level overview of the circuit that a program p is translated to.

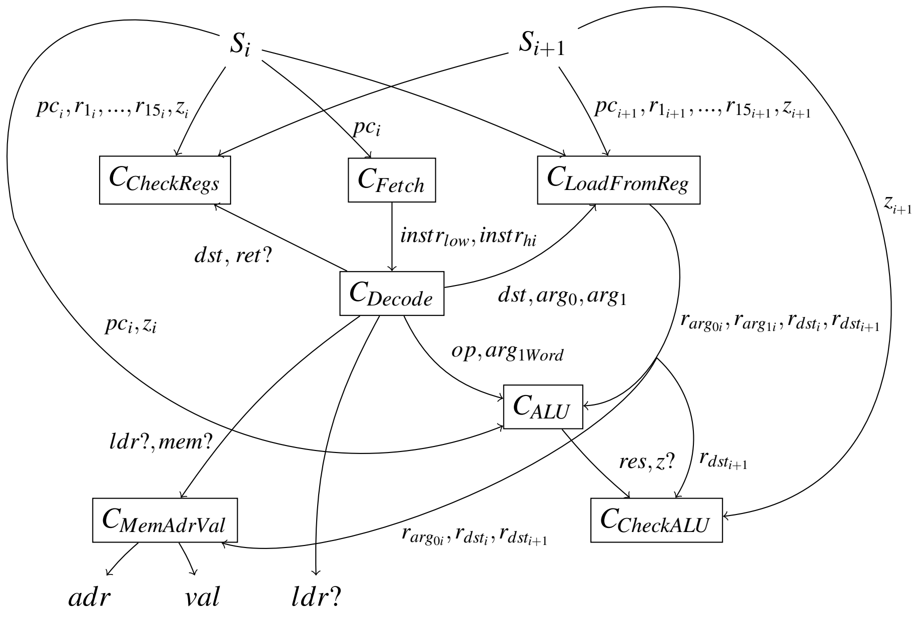


Figure B.2: The circuit C_{TR}

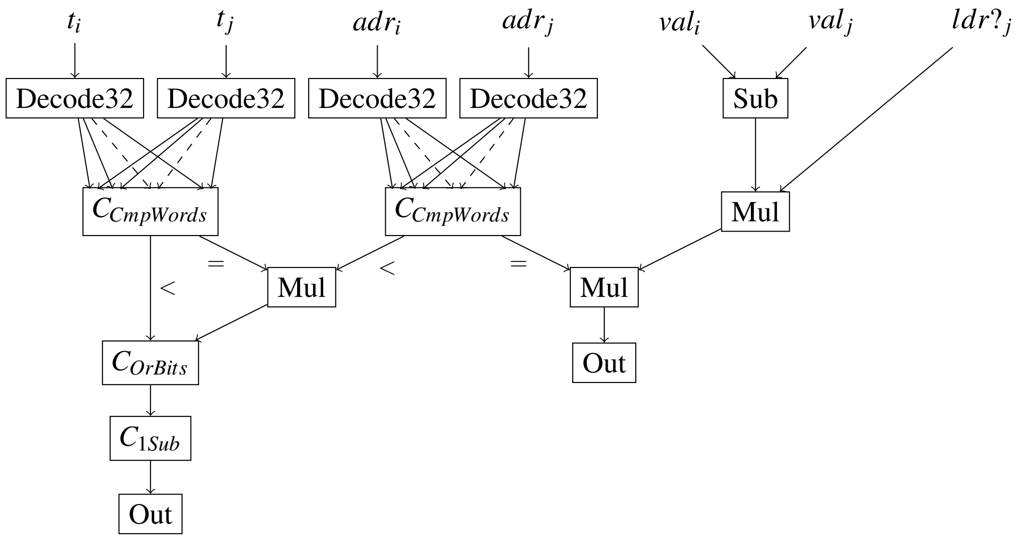


Figure B.3: The circuit C_{Mem}.

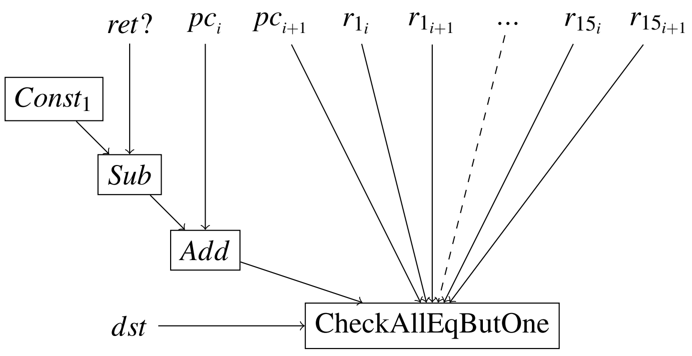


Figure B.4: The circuit $C_{CheckRegs}$

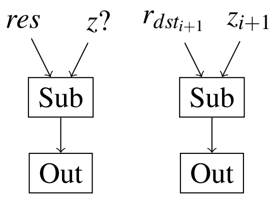


Figure B.5: The circuit $C_{CheckALU}$.

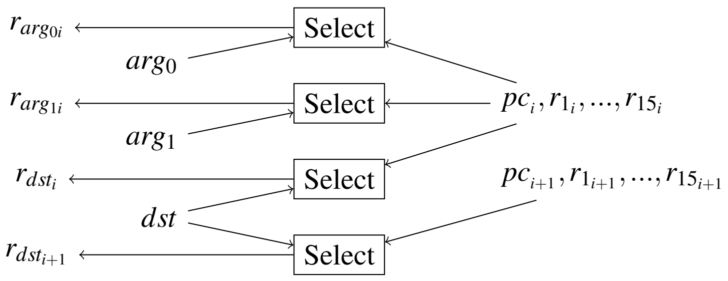


Figure B.6: The circuit $C_{LoadFromRegs}$.

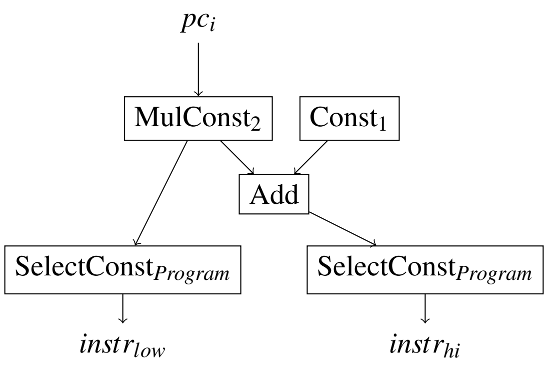


Figure B.7: Fetching the high and low word of instructions using two **SelectConst** gates, that are parameterized with constants resulting from encoding each instruction from the source program.

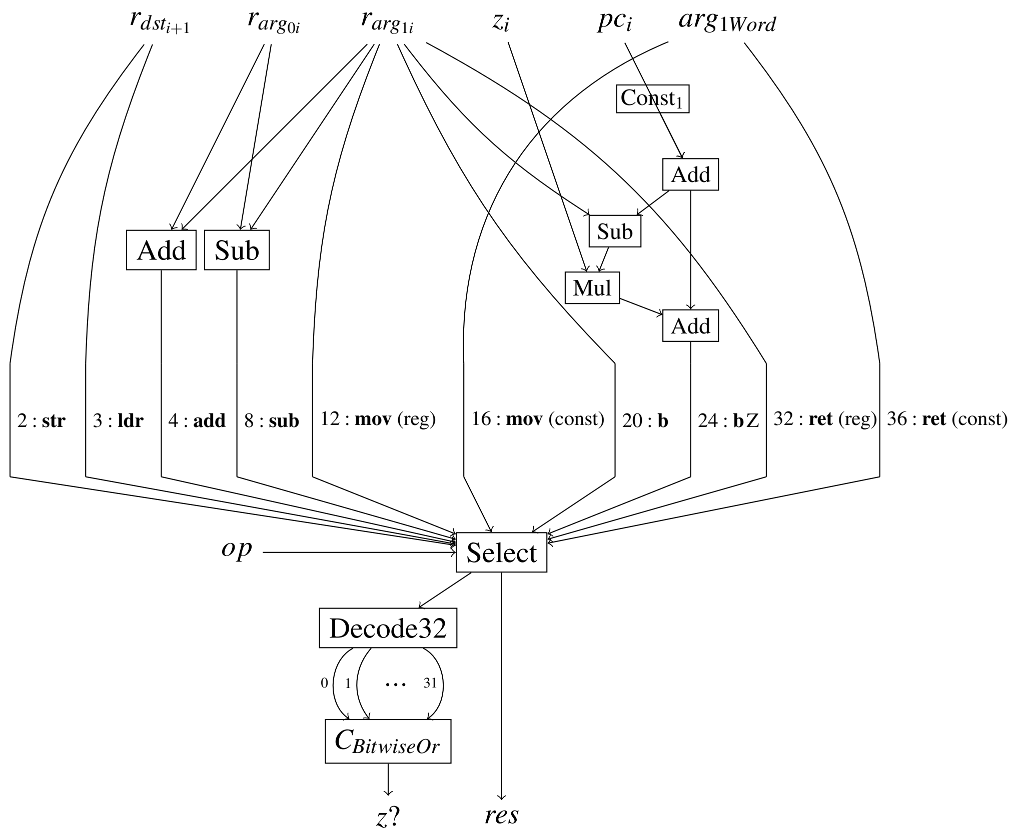


Figure B.8: The circuit C_{ALU}.

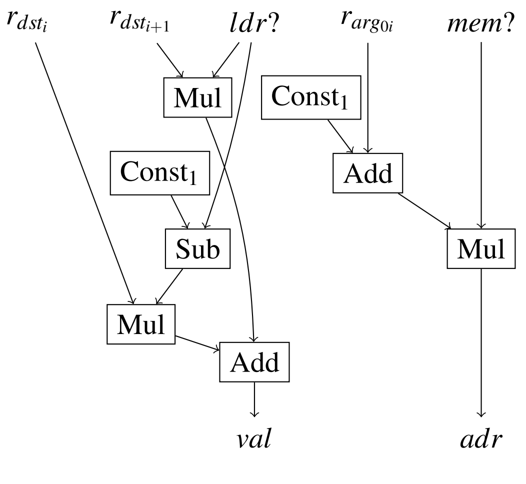


Figure B.9: The circuit $C_{CheckMemAdrValid}$.