

Formal Verification of Circuit-Soundness in zkWasm, a General Purpose zkVM

Vilhelm Sjöberg
CertiK
New York City, USA
vilhelm.sjoberg@certik.com

Roger Bai
CertiK
New York City, USA
rogerbai92@gmail.com

Hao Chen
CertiK
New York City, USA
hao.chen@certik.com

Xifeng Jin
CertiK
New York City, USA
xifeng.jin@certik.com

Zhaozhong Ni*
NVIDIA
Santa Clara, USA
zhaozhongn@nvidia.com

Sinka Gao
Delphinus Lab pty ltd
Sydney, Australia
xgao@zoyoe.com

Ronghui Gu
Columbia University
New York City, USA
ronghui.gu@columbia.edu

Zhong Shao
Yale University
New Haven, USA
zhong.shao@yale.edu

Abstract

Recent advances in zero-knowledge (zk) provers make it possible to implement an entire bytecode instruction set so that one can compile and run arbitrary programs and generate zk-proofs of the result. This is a novel kind of programming language implementation. Many blockchain applications are now being re-designed to improve privacy and throughput by verifying zk-proofs rather than running program logic directly. However, zk programming is famously error-prone, and in the blockchain setting, a simple error can lead to hundreds of millions of dollars of cryptocurrency being stolen in an undetectable way. Hence, it is crucial to ensure that a zero-knowledge virtual machine (zkVM) is correct.

We prove circuit-soundness for zkWasm, a state-of-the-art zkVM for the WebAssembly language. This is the first mechanized correctness proof of the complete circuits of a general-purpose zkVM, including the memory system. Compared to a conventional interpreter, the way a zkVM represents dynamically sized data structures requires a new set of lemmas. In order to verify the code, we use a hand-translated shallow embedding into Rocq, and this approach is also applicable to other zk-applications written using Halo2 or other PLONKish systems.

ZkWasm is a full-scale implementation being used in production systems, not just a prototype, yet only a small kernel needs to be verified, and the proof effort was reasonable. In the process, we uncovered three critical bugs, which illustrate the value of verification in this application domain.

*Work done at CertiK.



This work is licensed under a Creative Commons Attribution 4.0 International License.

CCS '26, The Hague, Netherlands

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846740

CCS Concepts

• **Theory of computation** → **Program verification**; *Cryptographic protocols*; • **Software and its engineering** → *Runtime environments.*

Keywords

WebAssembly, ZKSNARK, virtual machine

ACM Reference Format:
Vilhelm Sjöberg, Roger Bai, Hao Chen, Xifeng Jin, Zhaozhong Ni, Sinka Gao, Ronghui Gu, and Zhong Shao. 2026. Formal Verification of Circuit-Soundness in zkWasm, a General Purpose zkVM. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846740

1 Introduction

A *zero-knowledge proof of knowledge* is a construction that lets one party (the prover) convince another party (the verifier) that the prover possesses some private data (a witness) satisfying a certain set of constraints (the circuits). The construction is *sound* if the probability that the verifier accepts an incorrect zk-proof is negligibly small, *succinct* if the zk-proof object is small compared to the private data, and *zero-knowledge* if the zk-proof object does not leak any information about the private data except that it exists.

Over the last decade, zk-proofs have turned from a theoretical subject to a practical technology. For example, Galois [20] increased their ZKP performance by 10 times over a 2-year span. Jolt [22] claims to be able to increase zero-knowledge virtual machine performance by 20 times. Such improvements will enable zk-proofs to be used in computations ranging from Web3 to traditional scenarios such as decentralized identities, voting systems, internet of things, and supply chains [10].

For blockchains in particular, they solve two key problems. First, zk-proofs allow for *privacy*. In traditional blockchains like Bitcoin and Ethereum all payments are publicly visible, but Zerocash [4]

or Tornado.cash lets users receive money by submitting a zk-proof that there exists a transfer to them without revealing the sender.

Even more importantly, they can improve blockchain *throughput*. Currently, blockchain nodes process each individual transaction, but they are evolving towards a design where all business logic is computed by third-party computers that produce zk-proofs. The "zero-knowledge" property is not actually needed for this; the zk systems are used for their succinctness, which enables a "rollup" contract to quickly confirm the end-state of an execution history of thousands of transactions. The current roadmap of Ethereum relies on zk-rollups for its scaling [8].

However, this new reality presents a large correctness challenge. Ordinary smart contracts are written straightforwardly—they use a few assignments to update to the new contract state. For such code, manual code review can provide sufficient assurance. The constraint languages of zk-systems are much more difficult. First, everything must be expressed as hard-to-read polynomial equations over finite fields. Second, instead of computing the new state, the constraints must check the validity of an arbitrary witness chosen by an attacker.

One example illustrates how this is problematic: because of a one-character typo in a supporting library, the Tornado.cash circuit to compute hash functions failed to actually check the result and would accept any number as a valid hash of any string. In a normal program, such a simple error would be instantly detected, but here, the system faithfully processed transactions (containing correct hash values) without anybody noticing that a malicious transaction would be accepted too [9].

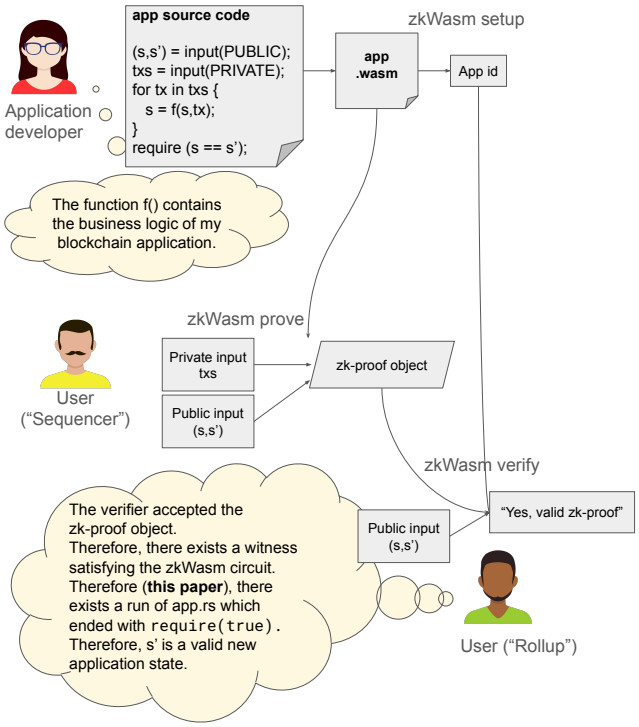


Figure 1: zkWasm usage

1.1 Zero-knowledge virtual machines

The first generation of zk applications worked by hard-coding the business logic as circuits. For example, one can implement a payment system by constraints to enforce that each transaction corresponds to a valid balance transfer. But lately, improvements in prover performance have enabled a new generation of applications based on general-purpose *zero-knowledge virtual machines* (zkVMs). These systems encode the semantics of a bytecode instruction set, so programs written in ordinary programming languages can be compiled using a standard toolchain to run on them. For blockchains, this means one can make the business logic programmable. For Ethereum, in particular, the usage of zero-knowledge Ethereum VM (zkEVM) is becoming a reality and is key to its future scaling [7].

ZkVMs offer backwards-compatibility because they can run smart contracts compiled from conventional programming languages. And they can enhance security because instead of implementing functionalities as zk-circuits, one can write the application as a normal program and limit the difficult zk-circuit programming to only the kernel that checks execution traces.

The subject of this paper, Delphinus Lab zkWasm [13], is a representative example of zkVM. It handles programs written in WebAssembly (Wasm), a general-purpose intermediate language targeted by compilers for many high-level languages, and it supports almost all Wasm except floating-point operations. The workflow for zkWasm usage is as follows (see Figure 1):

(1) The application developer writes a program and compiles it to Wasm. The program reads bytes using library functions `input(PUBLIC)` and `input(PRIVATE)`, and throws an error with `require` if some condition is not satisfied. The developer runs zkWasm on the program once to generate a verification key (a.k.a. an application id).
For a blockchain rollup, the Wasm program contains the business logic of some application. The private input txs is a long list of requests to the application signed by different users (e.g. requests to transfer funds between accounts), while the public input (s, s') consists of a representation of the current state of the application and a new state. The program checks the signature of each request, executes them in turn, and finally checks that the application state at the end is equal to the provided new state s'.
(2) Now a user can run the zkWasm *prover* specifying the wasm program and a set of public and private inputs. This will execute the program on those inputs, and if it runs to completion without errors, the prover will generate a zk-proof object.
For the blockchain scenario, this user is called the "Sequencer". He receives many requests from different users, and after collecting a large batch, he computes the new state $s' = f(s, txs)$ and runs the zkWasm prover on the triple (s, txs, s') to get a zk-proof object p.
(3) Finally, a different user can run the zkWasm *verifier*, specifying the application id, inputs, and zk-proof object. The

verifier will only accept the triple if there exists a run of the program with those inputs that does not fail any 'require' statement.
For a rollup, this part happens on the blockchain nodes. The Sequencer submits $((s, s'), p)$ to an on-chain smart contract, and if the verification succeeds, the contract updates the application state from s to s'. The point is that the chain only needs to check the small zk-proof p, not the long list of requests txs.

Is this secure? In our threat model, the Sequencer in Figure 1 is replaced with a malicious Attacker, who can submit anything to the Rollup and tries to make the application misbehave. The steps where this can go wrong are (a) There is a bug in the application f(), as in any other blockchain program. (b) There is a bug in the compiler, so the compiled wasm code is faulty. (c) There is a bug in the zkWasm constraints, so *verify* accepts a witness which satisfies the constraints but doesn't correspond to an execution of the application. (d) There is a bug in the underlying zk-proof verifier, so *verify* accepts a witness that doesn't satisfy the constraints.

In this work, we deal with part (c). This is a very challenging security problem, because in addition to the challenges of zk-circuit programming, the zkWasm implementation also has the large threat surface of a bytecode interpreter—the attacker can use a Turing-complete language to set up exploits. As we show in Section 5.1, an error in a single constraint line can let an attacker corrupt the program execution to make a load-instruction return a nearly arbitrary value; or a subtle design error can let the attacker make the program jump to an arbitrary address using an elaborate construction which would be nearly impossible to find by testing.

1.2 Correctness property

To deal with (c) above, we must formally verify that the constraints were implemented correctly. But what does "correct" mean? As we mentioned above, for zk proof systems there are standard properties of *soundness* and *zero-knowledge*. We can consider analogous properties for the zkVM, one level higher in the software stack.

A zkVM is *zero-knowledge* if it does not leak any information about the private inputs in Figure 1 except that there exists *some* input that results in a valid run. However, we get this by construction: a zkVM works by translating all inputs and the trace of the execution into a zk-witness, so as long as the zk-prover hides the content of the witness no information can leak out.

The interesting property is *soundness*, which says that if there is a witness satisfying the zkVM constraints, then there exists a run of the program. Together with soundness of the underlying zk-prover, this implies that if the verifier accepts the zk-proof object, then there exists a run of the program. The bugs that make zk-applications so difficult are precisely the ones that violate soundness: the programmer made a mistake when writing the constraints and (like Tornado.cash) the application accepts a bogus witness constructed by the attacker.

Soundness is also analogous to the usual correctness property for programming language compilers [17] which says that for every run of the compiled program there exists a run of the source code. But as we shall see (Section 3.2) our statement will be slightly different because we are given a particular witness and because zkVMs can receive private inputs.

Dually, one can also consider *completeness*: that every run of the program can be encoded as a valid witness. This property would rule out bugs that prevent the system from executing a program (i.e., bugs that affect the *liveness* of a blockchain). For example, if the programmer made a mistake and wrote an unsatisfiable constraint $x = x + 1$, then no proof object would ever be accepted and soundness would be vacuously true. Actually, for completeness it is not enough to verify only the constraints: a bug in the surrounding code that runs the program and constructs a witness could also make the system fail to execute a program. However, this class of bugs is not as critical a concern as soundness. First, we only care about the legitimate users being able to run their programs, while for soundness we need to handle inputs constructed by a malicious attacker. Second, completeness is amenable to normal software testing techniques, as for a conventional compiler or bytecode interpreter. Soundness addresses the specific thing that makes zk software so challenging, namely the possibility of missing constraints.

Therefore, soundness of the constraints is the key remaining ingredient to ensuring a zkVM is secure. In the rest of the paper we will take that as the goal to ensure correctness.

1.3 Contributions

In this paper, we successfully apply formal verification to prove soundness of the entire constraint system for a fully-featured zkVM. We start by hand-translating the zkWasm constraint code from Rust into parameters and axioms in Rocq. (In future work, we will want to compile the axioms into executable code to not need to trust the manual translation.) We define a small-step operational semantics for the Wasmi bytecode. We then prove an upwards simulation:

> **Theorem.** If there exists a witness satisfying the zkWasm constraints, then there exists a Wasmi execution without errors starting from the initial state.

This theorem rules out any kind of soundness/underconstrainedness errors in the constraints. Although the Attacker can submit maliciously constructed witnesses, the only ones that are accepted are encodings of real program runs.

The paper starts by describing the background to this work (Section 2). We assume no prior knowledge about zk-systems, explain the API that is provided by a zk-prover, and use zkWasm as a running example to show how to use that API to implement a VM.

Then we formalize in Rocq how zkWasm works by a small-step semantics of the input language Wasmi (Section 2.3), which we adapt from existing work on the semantics of Wasm, and a refinement relation from the abstract language state to the zkWasm tables (Section 3.2). We state the correctness theorem, by adapting the usual correctness statement for interpreters/compilers to the zkVM setting (Section 3.2)—in particular, we include a "knowledge extractor" to specify the knowledge of the prover.

Then we go on to describe the correctness proofs (Section 4). The style of proof we create can serve as a blueprint for other efforts to verify zkVMs. Most importantly, we verify the most distinctive feature of a zkVM, the memory table:

- ZkWasm uses a classic idea from the zkVM literature to represent dynamic data by sorted lists of operations. It uses two slight variations on this to represent the memory and the call stack. Prior to our work, nobody had published a detailed proof of correctness of this approach, perhaps because they considered it intuitively obvious, but we develop a sequence of somewhat tricky lemmas to prove it correct (Section 4.1.3), and in doing so we found an error in one of the two tables.
- In addition to correctness, our approach to the memory table has an important software-engineering aspect. We specify the table in a way that makes explicit the programmer's intuition that it represents a mutable memory (Section 4.1.2). This separates the proof of the memory subsystem from the proofs of the individual instructions, so that the verification effort can be divided into manageable subtasks (Section 4.1.4).

Any future verified zkVMs will need to deal with the memory subsystem, and will benefit from adopting our approach to keep the implementation details from complicating the rest of the proofs.

The practical details of our proof are also of interest to other practitioners:

- We use the fact that zkWasm uses a modern "PLONKish" constraint language and is written in a "direct" style to create a *shallow* embedding of the zk-constraints (Section 3.1), which is easier to work with in the proof assistant than deep embeddings or combined models of Rust and the zk-constraints. This helps make our large development (tens of thousands of lines of proof script) economically feasible. The same approach will be suitable for any zk-application written in the same style.
  - One feature of our embedding is that we can selectively add an assumption that certain numbers never overflow. So some overflow proofs were only done by manual inspection, but not formalized in Rocq. This tradeoff also reduces the verification effort and helps us tackle larger code (Section 3.1).
- As an experiment, we port the proof of one of the zkWasm modules to Verus, an SMT-based prover, and thereby cut the proving effort to less than half (Section 5.3). This shows that our proof approach is general and not limited to the particular tools we used.

Finally the paper evaluates our experience in term of effort (Section 5.2) and the bugs we found (Section 5.1). Some of our experience should be of interest to zkVM designers:

- We report which part of the zkVM were easy versus labor-intensive to verify (Sections 4.2–4.3). This is quite different from a conventional VM, for example non-aligned memory instructions and 64-bit arithmetic make up a big part of the effort.
- The three bugs we discovered are examples of the kind of errors that can occur in a zkVM. They illustrate how even small slips create critical security vulnerabilities.

2 Background and Overview

Before describing our formal verification, we explain how zkVMs in general, and zkWasm in particular, are implemented. To see why it is challenging to verify, we can draw a comparison to verifying a conventional bytecode interpreter. The witness in zkWasm is a set of tables whose contents satisfy a set of constraints; the most important table is the ETable, which has a row for each step of program execution. The various parts of our proof can be analogized to their interpreter counterpart:

The constraints ↔ the interpreter source code.
"Each row of ETable corresponds to one step of the operational semantics" (our Theorem 3.2) ↔ "the step() function follows the operational semantics"
"If an ETable row has the OpBinBit set, the row is the result of a correctly executed bitwise instruction (Section 4.3)" ↔ "the function interpreting bitwise instructions is correct."
"Entries in the BitTable represent the results of bitwise operations" (Section 4.2) ↔ "The standard library functions for and/or/xor are correct."
"Entries in the MTable correspond to writes or reads in a memory" (Section 4.1) ↔ "the data structures that the interpreter uses to implement memory are correct".

From this comparison we can see three reasons the zkVM is harder to verify than an interpreter. First, instead of a function to compute the next row it has constraints, and we must prove that only a valid row can satisfy them. Second, the code is more complicated because the primitives exposed by the zk-prover are simpler. An interpreter can compute addition in a single statement (x+y), while as we will see below, zkWasm's ADD needs to emulate 64-bit arithmetic using field arithmetic. And in a conventional interpreter bitwise instructions and memory operations are one-liners, but in the zkVM they have a nontrivial implementation which we need to verify. Third, while the attacker can craft an entire table, the constraint language can only express *local* constraints, which relate only a few different rows at a time. We must use local constraints to prove the global invariants of, for example, the memory data structures.

Accordingly, in the remainder of this section we will explain the constraint language in more detail, and then illustrate how such constraints can execute an example program.

2.1 PLONKish constraints

ZkWasm's circuit is implemented using the front-end of the Halo2 proof system [25], which supports a constraint system format first pioneered by the PLONK zk-prover [12]. In this format, the witness consists of a two-dimensional table of numbers in a finite field $\mathbb{F}_p$. The circuit constraints are equations between numbers in the table. In zkWasm the witness table contains a few hundred columns and 2^k rows where the value $18 \leq k \leq 25$ is chosen at circuit-construction time. The programmer thinks of different groups of columns as defining different sub-tables. The field $\mathbb{F}_p$ comes from a carefully chosen elliptic curve with order $p \approx 2^{254}$, but for the purpose of verification, we can just think of the table entries as integers modulo some large prime number p.

The zk-circuit programmer can specify four types of constraints on the tables. **Equality constraints** state that two particular table cells are equal. **Fixed columns** have all the values provided at circuit-construction time. **Gates** are polynomial equations. The variables in the polynomial are cells specified by column and a small row offset. That is, the equation can refer to cells in the

"current row" as well as a handful of rows above and below it, and this equation must hold for every table row. **Lookups** specify two tuples of columns, $(c_1, \ldots c_n)$ and $(c'_1, \ldots c'_n)$, and are satisfied if $(c_1, \ldots c_n)$ is a foreign key for $(c'_1, \ldots c'_n)$.

The (considerable!) challenge of implementing a zkVM is to use such constraints to force the table entries to represent the data structures that arise during the execution of a Wasm program.

2.2 The zkWasm processing stages

ZkWasm relies on two pieces of third-party software: the Wasmi interpreter to execute Wasm programs, and Halo2 to generate zk-proof objects. The structure of the program is shown in Figure 2.

First, zkWasm calls Wasmi to translate the source program from Wasm to Wasmi bytecode, which is an equally expressive but somewhat more low-level language (e.g. with jumps instead of structured control flow). Then, zkWasm translates the bytecode into a Halo2 table called ImTable, which represents the program text and the initial values of global variables. Each Wasmi bytecode instruction is "packed" into a field element by multiplying the opcode and immediates with powers of 2 to shift them, and then adding them together. For example, if the lth line of the fth function is `CALL m`, then we assign some table row (with arbitrary number i) as $\mathsf{ImTable}(i) = f \times 2^{146} + l \times 2^{114} + \mathtt{CALL} \times 2^{128} + m \times 2^{96}$. In addition to ImTable this translation step also produces two auxiliary tables, namely a branch table used by the BrTable instruction and a call table used by the CallIndirect instruction.

ImTable and the branch tables are "fixed" tables, i.e., they form a part of the constraint system rather than the witness. This is the only part of the constraints that depends on the input Wasm program. The other constraints, the "circuits", were written once and for all by the zkWasm developer. (In this work, our goal is to verify the circuits.) To finish the "setup" step from Figure 2 zkWasm calls Halo2 to turn the ImTable as well as the other constraints into the verification key (a.k.a. application id).

Next, a user calls zkWasm to prove a particular program execution. It uses Wasmi to run the program with the given inputs and record a *trace*, a list of every instruction executed. ZkWasm translates the trace into a set of tables. The main *ETable* has a row for each step in the trace, and its columns contain the fixed-size state of the Wasm machine (the same kind of things that would be stored in registers in a conventional CPU). There are also auxiliary tables: *MTable* represents stack, heap, and globals; *JTable* represents the return addresses on the call stack; and *BitTable* has information related to each bitwise operation performed in the trace. Finally, zkWasm calls Halo2 with all the tables and constraints, to produce a zk-proof object that the data in the tables (the witness) satisfies the constraints.

2.3 The Wasmi bytecode language

The Wasmi bytecode that zkWasm operates on uses almost the same set of instructions as Wasm, but with some simplifications. Wasm's structured control flow (while, if) is replaced with a conventional program counter with branch instructions, the Wasm operand stack is represented with a stack pointer, and Wasm local variables are placed on the same operand stack. The result is a stack machine with three separate byte-addressed memory spaces for stack, globals, and memory, as well as a read-only instruction memory. The compiled Wasmi program has the exact same behavior as the original Wasm one, but zkWasm benefits from the simpler machine.

In order to specify the correctness of zkWasm, we define the semantics of Wasmi bytecode in Rocq. Happily there already exists a well-tested specification of Wasm in Rocq, WasmCert [24], which we rely on as much as we can to avoid specification errors. We define our own small-step rules, but re-use the auxiliary definitions from WasmCert for computing arithmetic operations, accessing memory, and accessing global variables.

Figure 3 shows syntax and semantics. There are 27 types of instructions, which mostly coincide with Wasm except for flow control. We define the high-level state σ of our abstract machine, which is a record with five components. The globals and memory are re-used from WasmCert, while the program counter pc and call stack reflect the more low-level semantics of Wasmi bytecode.

Our operational semantics for Wasmi has 47 step rules. In the figure we show a few representative examples. The semantics are defined with respect to some fixed program, and fetches the current program counter σ.pc. The Add rule computes the sum of the two top items on the stack (modulo 32 or 64, depending on the s flag), while the Xor rule computes their bitwise XOR. Note that we reuse the definition of `Wasm.xor` from WasmCert. Both rules increment the program counter and push the result to the stack. The rules for Br, Call, and Return show how control flow instructions work. The Branch instruction jumps to an immediate destination address l, while discarding d items from the stack but optionally keeping the top item x. Call and Return manipulate the callstack component of the state.

2.4 How zkWasm executes a program

ZkWasm translates the execution of the program into the main table, the *ETable*. As an example, we consider the program

```
int balance, amount;

void f() {
  balance = 100; amount = 10; balance -= amount;
}
```

This is compiled into Wasmi byte code, which (slightly simplified) becomes `CONST 100; STORE bal; CONST 10; STORE amt; LOAD bal; LOAD amt; SUB; STORE bal`, where `amt` and `bal` are addresses of heap variables. When executed the program takes 8 steps, which are translated into 8 rows of the ETable. The ETable contains a few hundred columns which represent the fixed-size state of the program (while dynamic-sized state like the stack and heap are in auxiliary tables). For our example program, we show a subset of the columns in Figure 4.

22 ETable columns are shared by all the different types of instructions. Among these, enabled is a bit which is set for the first n table rows (where n is the length of the trace); eid is a unique row number counting upwards $1, 2, 3, \ldots$; the pair (fid, iid) represents the current program counter (as function id and line number); opcode is some integer; and sp is an integer used as a stack pointer.

In addition to the shared columns, the ETable contains 27 additional groups of columns, one for each type of Wasm instruction. There are 27 "selector" columns called OpBin, OpCall, etc.

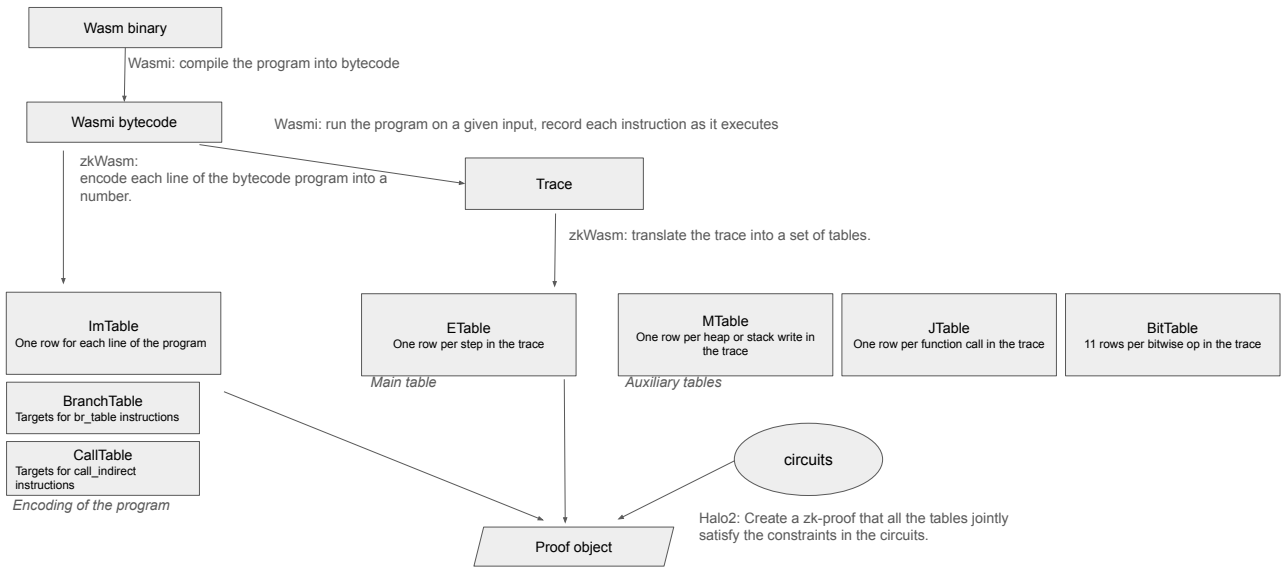


Figure 2: The structure of zkWasm. (This entire diagram represents the Prove step in Figure 1.) Our goal is the verify the "circuits", a set of constraints that enforce the semantics of Wasm program execution.

(Instructions)

$$
\begin{array}{rcl}
ls & := & \text{1BYTE} \mid \text{2BYTE} \mid \text{4BYTE} \mid \text{8BYTE} \quad s := \text{32bit} \mid \text{64bit} \quad k := \text{keep} \mid \text{nokeep} \quad \mathit{ConvOp} := \text{8to32} \mid \text{16to32} \mid \cdots \mid \text{32to64} \\
\mathit{BinOp} & := & \text{ADD} \mid \text{SUB} \mid \text{MUL} \mid \text{DIV}_u \mid \text{REM}_u \mid \text{DIV}_s \mid \text{REM}_s \quad \mathit{BitOp} := \text{AND} \mid \text{OR} \mid \text{XOR} \quad \mathit{SOp} := \text{SHL} \mid \text{SHR}_u \mid \text{SHR}_s \mid \text{ROTL} \mid \text{ROTR} \\
\mathit{UnOp} & := & \text{CTZ} \mid \text{CLZ} \mid \text{POPCNT} \quad \mathit{RelOp} := \text{EQ} \mid \text{NEQ} \mid \text{GT}_s \mid \text{GT}_u \mid \text{GE}_s \mid \text{GE}_u \mid \text{LT}_s \mid \text{LT}_u \mid \text{LE}_s \mid \text{LE}_u \\
I & ::= & \text{Bin}_s\ \mathit{BinOp} \mid \text{BinBit}_s\ \mathit{BitOp} \mid \text{BinShift}_s\ \mathit{SOp} \mid \text{Unary}_s\ \mathit{UnOp} \mid \text{Rel}_s\ \mathit{RelOp} \mid \text{Conv}_s\ \mathit{ConvOp} \\
 & & \mid \text{Br}_k^d \mathit{Int}_{64} \mid \text{BrIf}_k^d \mathit{Int}_{64} \mid \text{BrIfEqz}_k^d \mathit{Int}_{64} \mid \text{BrTable} \mid \text{Call}\ \mathit{Int}_{32} \mid \text{CallIndirect}\ \mathit{Int}_{32} \mid \text{CallHost} \mid \text{Return} \\
 & & \mid \text{Const}_s\ \mathit{Int}_{64} \mid \text{Drop} \mid \text{Select} \mid \text{Test}_s \mid \text{GlobalGet}\ \mathit{Int}_{32} \mid \text{GlobalSet}\ \mathit{Int}_{32} \mid \text{LocalGet}_s\ \mathit{Int}_{32} \mid \text{LocalGet}_s\ \mathit{Int}_{32} \mid \text{LocalTee}_s\ \mathit{Int}_{32} \\
 & & \mid \text{Load}_{s,ls}\ \mathit{Int}_{64} \mid \text{Store}_{s,ls}\ \mathit{Int}_{64} \mid \text{MemorySize} \mid \text{MemoryGrow}
\end{array}
$$

(Program state)

$$
\begin{array}{rcl}
\mathit{label} & := & \mathbb{Z} \times \mathbb{Z} \quad \text{(Function id (fid), and instruction id (iid) within the function.)} \\
\sigma & := & \{\text{pc} : \mathit{label};\quad \text{stack} : \text{list}\ \mathbb{Z};\quad \text{callstack} : \text{list}\ \mathit{label};\quad \text{memory} : \text{Wasm.memory};\quad \text{globals} : \text{list Wasm.global}\}
\end{array}
$$

(Example step rules)

$$\frac{\mathit{program}[\sigma.\text{pc}] = \text{Call}\ f \qquad \sigma.\text{callstack} = s}{\sigma \to \sigma\{\text{pc} := (f, 0)\}\{\text{callstack} := (\sigma.\text{fid}, \sigma.\text{iid} + 1) :: s\}}$$

$$\frac{\mathit{program}[\sigma.\text{pc}] = \text{Br}_{\text{keep}}^d\ l \qquad \sigma.\text{stack} = x :: xs +\!\!+ ys \qquad \text{length}(xs) = d}{\sigma \to \sigma\{\text{pc} := l\}\{\text{stack} := x :: ys\}}$$

$$\frac{\mathit{program}[\sigma.\text{pc}] = \text{Return} \qquad \sigma.\text{callstack} = l :: s}{\sigma \to \sigma\{\text{pc} := l\}\{\text{callstack} := s\}}$$

$$\frac{\mathit{program}[\sigma.\text{pc}] = \text{BinBit}_s\ \text{XOR} \qquad \sigma.\text{stack} = x_r :: x_l :: xs}{\sigma \to \sigma\{\text{iid} := \sigma.\text{iid} + 1\}\{\text{stack} := (x_l\ \text{Wasm.xor}\ x_r) :: xs\}}$$

$$\frac{\mathit{program}[\sigma.\text{pc}] = \text{Bin}_s\ \text{ADD} \qquad \sigma.\text{stack} = x_r :: x_l :: xs}{\sigma \to \sigma\{\text{iid} := \sigma.\text{iid} + 1\}\{\text{stack} := ((x_l + x_r) \mod 2^{(1+s)*32}) :: xs\}}$$

Figure 3: Wasmi syntax and operational semantics

which control which group is "active". The selector columns are constrained to contain only 1 or 0, and to sum to 1 so that exactly one of the bits is set.

In the attack scenario, the contents of the ETable is under the control of the attacker, and the role of the constraints is to make sure that the only way to fill it in is with a valid program trace. Consider for example the stack pointer sp. The stack grows downwards, so when the CONST instructions adds one item to the stack, the sp should be decremented, e.g. from 333 to 332 in the example. Accordingly there is a gate constraint:

$$\text{ETable}(\text{OpConst}, i) \times (\text{ETable}(\text{sp}, i + 1) - (\text{ETable}(\text{sp}, i) - 1)) = 0$$

enabled	eid	fid	iid	sp	opcode	···	OpLoad	value	OpStore	value	OpBin	op	lhs	rhs	res	OpConst
1	1	7	0	333	CONST 100		0		0		0					1
1	2	7	1	332	STORE bal		0		1	100	0					0
1	3	7	2	333	CONST 10		0		0		0					1
1	4	7	3	332	STORE amt		0		1	10	0					0
1	5	7	4	333	LOAD bal		1	100	0		0					0
1	6	7	5	332	LOAD amt		1	10	0		0					0
1	7	7	6	331	SUB		0		0		1	SUB	100	10	90	0
1	8	7	7	332	STORE bal		0		1	90	0					0

Figure 4: Example ETable

If the OpConst bit is zero this equation is vacuously true, but if the bit is active it forces $\mathsf{ETable}(\mathsf{sp}, i+1) = \mathsf{ETable}(\mathsf{sp}, i) - 1$. In addition to this constraint there is also a lookup-constraint forcing the value to be written to the stack (to be discussed later). There is an equation making sure that if OpConst is set then the opcode is indeed CONST. And there is a lookup constraint enforcing that for every row in the ETable, the ImTable contains a value encoding the tuple (fid, iid, opcode); this constraint is the zkVM version of an instruction fetch.

Arithmetic instructions. Now consider binary arithmetic such as the SUB instruction. These instructions use some "scratch space" columns in the ETable, including the columns lhs, rhs and res which read/write to the appropriate stack locations.

In our example we see that lhs = 100, rhs = 10, res = 90, so the instruction correctly computed a subtraction. But enforcing this is not as easy as writing a constraint $\mathsf{ETable}(\mathsf{res}, i) = \mathsf{ETable}(\mathsf{lhs}, i) - \mathsf{ETable}(\mathsf{rhs}, i)$. The table elements are in the finite field, modulo a large integer p, while in Wasm programs we want to compute the arithmetic modulo 2^{64}.

Instead, the column group for the arithmetic instruction also includes some columns like modulus (= 2^{64} for 64-bit subtractions), overflow, etc, and more complicated equations like

$$\mathsf{OpBin} \times \mathsf{isSub} \times (\mathsf{rhs} + \mathsf{res} - \mathsf{lhs} - \mathsf{overflow} \times \mathsf{modulus}) = 0$$

and we must verify that these equations imply that the arithmetic works correctly.

Bitwise instructions. What if instead of SUB the program had XOR? This is even more difficult: there is no way to use polynomial equations to force res = lhs XOR rhs. Instead zkWasm uses an auxillary table called BitTable. Each bitwise operation that occurs in the program generates 11 rows in the BitTable: 8 of the rows are bitwise operations applied to single bytes (where we can look up the answer in a hardcoded table), and there are constraints forcing the other 3 rows to be shifts and adds of the rows below them, to gradually build up the correct answer. The zkWasm implementation of the XOR instruction then has a lookup constraint saying that there exists a row $(1, \mathsf{XOR}, \mathsf{read_lhs}, \mathsf{read_rhs}, \mathsf{res}) \in \mathsf{BitTable}$.

In our development, to prove that the XOR implementation is correct, we first have to prove a lemma that every row in the BitTable contains a correct result of a bitwise operation. In general, we need to prove correctness lemmas for all the auxillary tables in order to finally prove the ETable correct.

start_eid	end_eid	type	address	value
4	100	heap	amt	10
2	8	heap	bal	100
8	100	heap	bal	90

Figure 5: Example MTable

Memory instructions. The area where a zkVM differs the most from a conventional interpreter is how it represents dynamically-sized state, like the stack and heap. In zkWasm this is done using an auxiliary table called MTable, which contains the history of all memory writes in the entire trace lexicographically sorted by memory address, then time of access. For our example program, the entries related to the LOAD and STORE instructions are as in Figure 5. (The full table also has entries related to stack accesses, marked "stack" instead of "heap", which we omit for simplicity.)

The start_eid and $\mathsf{end_eid}$ columns in MTable refer back to the eid column in ETable. The start EID is the EID of the step in the trace that wrote that value to the address, and the end EID is the step that will next write to that address (or some large value like 100 if the address is never written to again).

The implementation of the STORE instruction uses a lookup constraint to ensure there is a row in MTable with start_eid = eid. And the LOAD instruction has lookup constraints enforcing that there is some row with $\mathsf{start_eid} \le \mathsf{eid} < \mathsf{end_eid}$.

The more difficult part is constraining MTable so that it really represents a valid trace with no spurious extra rows (a global property). To this end it has an additional column mops which we will discuss later, in Section 4.1.

2.5 Verifying constraints versus verifying constraint-construction code

So far, we have discussed the constraints as mathematical formulas, but let us consider what they look like in the actual zkWasm source code. ZkWasm is written in Rust and uses the API provided by Halo2 to build up the constraint formulas. Names of columns are stored in normal variables, and by operator overloading it is possible to write formulas using nearly normal notation (but passing around a "meta" variable to keep a reference to the constraint system). Thus, for example, part of the circuit for the Xor instruction says that in each row, the column called `bit_table_lookup_result` should

equal the column called `res`. The gate construction operator takes a list of polynomials, which are all asserted to equal zero.

```
constraint_builder.push("op_bin_bit: lookup",
 Box::new(move |meta| {
  vec![bit_table_lookup.result.expr(meta) - res.expr(meta)]}));
```

But not every constraint is represented so directly. Since the system is written in Rust, it is possible to use helper functions and Rust macros to create multiple constraints programmatically. In particular, rather than using the Halo2 table interface directly, zkWasm uses about 1000 lines of Rust code which it calls "allocators" to manage the columns of the ETable and MTable. The constraints are written in terms of several "logical" tables and columns, which are interleaved into the actual Halo2 witness table. While Halo2's tables just contain numbers modulo p, the allocator columns can be typed as bools, 32- or 64-bit integers, or "common range" numbers ($0 \leq x < 2^k$). It is the responsibility of the allocator to enforce that the numbers are in the right range. Thus, for example, the code to construct the ETable constraints declares a column `eid = allocator.alloc_u32_state_cell()`. The circuit programmer writes equations using `eid` as a variable, but behind the scenes the allocator code added four additional columns for the bytes of the word, and arithmetic and lookup constraints to force `eid` to range over 32-bit numbers.

Our guiding principle is that verification of the constraints and verification of the constraint-generating code are two separate problems. Once the constraint system has been imported into a proof assistant, one needs to prove that each table satisfies the intended invariants, and that they imply the final soundness properties. That is the problem we solve in this paper, and it is the same for all PLONKish zk-provers. On the other hand, reasoning about the constraint-generation is the familiar task of verifying general-purpose programs. For this paper, therefore, we decided to verify the constraints, but trust the correctness of the allocator code.

3 The Verification Task

Having explained the zkWasm system, we now describe how we modeled it in the Rocq proof assistant and the theorem we prove about it. The material in this section is specificational, i.e. a mistake in these definitions would mean we prove the wrong theorem, as opposed to the proofs (Section 4) which are machine-checked.

3.1 Machine model in Rocq

Before we can prove anything about the zkWasm circuits, we must first translate them into the proof assistant, which we did by hand. Our approach is a shallow embedding into Rocq. Each of the tables are postulated as `Parameters`, e.g. for the BitTable we assume there is a function `bit_table_values` which maps each (column, row) pair to the contents of a cell in the bit_table. The circuits from zkWasm are then translated to `Axioms` about properties that the table satisfies. We call the combination of parameters and axioms the "model", because it represents the code of zkWasm inside Rocq.

This kind of shallow embedding is ideal for proving a particular zk-application like a zkVM. By contrast, something like a verified compiler into Halo2 (e.g. [23]) needs a deep embedding, where there are Rocq datatypes for Halo2 tables and Halo2 constraints. In principle, all the theorems of this paper can be proven with either a deep or a shallow embedding. However, in practice it makes a big difference to the proof effort: it is much better to just work directly with native Rocq statements about integers, than to have to first unfold the embedding for each line of constraint code. In our experience with other projects this could make up to a 2x difference in speed of proving, which is crucial for a big project like this. Therefore, we view the "direct" style of zkWasm (Section 2.5) as a decisive advantage for formal verification.

Translating gates. We manually translate the circuits by writing Rocq code that corresponds to the Rust source (making no phase distinction between the functions and macros). Thus for example, the zkWasm Rust code fragment

```
vec![( lhs.u64_cell.expr(meta) + rhs.u64_cell.expr(meta)
    - res.expr(meta)
    - overflow.expr(meta) * size_modulus.expr(meta))
  * is_add.expr(meta)]
```

gets translated into a similar Rocq expression

```
Axiom bin_add : pgate etable (fun get ⇒
 (get lhs_u64_cell 0 + get rhs_u64_cell 0 - get res 0
- get overflow 0 * get size_modulus 0)*( get is_add 0) :: nil).
```

The Rocq definition `pgate` expands out into a statement that the list of formulas evaluate to 0 at each row of the table. We keep the Rocq code line-by-line similar to the original code to minimize the risk of translation mistakes, which works because zkWasm is written in a "direct" style.

Cell Allocation and Range Axioms. For the ETable and MTable, zkWasm uses the "allocator" approach described in Section 2.5 and each "logical" column in the Rust code is treated as a table column in the Rocq code when we translate the gates. Since we trust the allocation code, the range constraints from the types of the columns are modeled as axioms.

Integer overflow. In many of the proofs we decided to not handle the possibility of overflows in the field arithmetic, i.e., the numbers are treated as integers rather than integers modulo p. This is because Rocq's proof automation works much better for goals that do not include the modulo operation (they can be solved instantly by the "linear arithmetic" tactic, whereas for expressions with modulo the user needs to do step-by-step algebra, which takes much more effort). It is clear from manual inspection that the values in many of the gates are so small that they will not overflow, so adding this as a trusted assumption is a worthwhile trade-off to save proof effort. The basic idea is that if x, y and z are 64-bit unsigned integers, then the equations $(x+y+z) \mod p = 0$ and $x+y+z = 0$ have the same solutions. Similarly $(x+y-z) \mod p = 0$ and $(x+y-z) = 0$ have the same solution if x and y are 64-bit. The human auditor needed to review the zkWasm equations to check that they have enough type constraints. (Ideally in future work this reasoning should also be formalized.)

We took care to model the modulo operation precisely in the cases we thought it might be unclear: for all the "encoding" functions that pack multiple values into one integer, and for the arithmetic instructions, since math operations require being particularly careful about overflow. To express the difference, we use two definitions: `gate` (which assumes the gate expression values are small)

versus pgate (which forces the proof engineer to prove the non-overflow).

Trusted Computing Base. Our approach entails the following trusted components: The manually translated constraints must correctly correspond to the original zkWasm constraints written in Rust. In the cases where we elected to not include the modulo operation in the gate, we must have correctly assessed that there is no way to make the numbers overflow. The zkWasm allocator code must be correct. Our Wasmi specification must represent the intended semantics of Wasm. The zkWasm code to translate the Wasm program and to represent the initial state (including initial values of global variables and the public/private inputs) must be correct, since it is outside the scope of our verification. We trust that Halo2, the Wasmi frontend (including type validator), and of course Rocq, are implemented correctly.

Is hand-translation good enough? All in all we translate around 5000 lines of zkWasm constraints into Rocq axioms. This entails some risk because zk-programming is famously error prone. We could make a mistake when translating an equation, and if that line had a security bug we might accidentally fix the bug, so the verification would miss it.

In practice, we think the risk is fairly small. In our usual approach at CertiK we combine formal verification with conventional code auditing, because auditing is cheaper, can cover a larger codebase, and can discover problems outside the formal correctness statement. Then simple typos can be caught by the audit, while the formal verification is crucial for subtler logic errors. Indeed, the first of the three bugs in Section 5.1 we noticed already while translating the constraints, before starting the verification proper.

In future work, to get an end-to-end proof the constraints should be imported in a mechanized way. One approach is what Nethermind did (related work reference [18]), to instrument the zk-backend to pretty-print the constraint-expressions and prove the pretty-printed constraints imply the hand-written ones. Alternatively one might use a tool like Aeneas [14] or rocq-of-rust [11] to import the Halo2 API calls into Rocq and prove what the list of constraints is. Either way, once the constraints are represented in the proof assistant, one is left with the task we deal with in this paper, to prove that the zkVM data structures and instruction implementations are correct.

3.2 Top-level specification

Our ultimate correctness property is similar to a verified compiler: an upwards-simulation [17], saying that every step of the implementation can be simulated by a step of the specification. However, instead of proving that every execution of a compiled program can be simulated, here we are given a particular purported execution (encoded in the ETable). An implementation "step" means going to the next ETable row.

Simulation relation. We define (Figure 6, which represents about 90 lines of Rocq) a relation $R\ i\ \sigma$ between ETable rows and abstract states, saying that the ith row represents the Wasmi state σ. The definition uses a function memory_at *eid type*, which scans through the MTable and returns a finite map $\mathbb{Z} \to \mathbb{Z}$ that represents the memory state at step number *eid* (see Section 4.1).

The definition of R has separate parts for the different parts of σ. The **program counter** is stored directly in ETable. For the **stack** we define a relation $R_{\text{stack}}\ eid\ sp\ xs$, which says that the list xs is stored in stack memory starting at address sp. The ETable column $\text{ETable}(\text{sp}, i)$ stores the current value of the stack pointer at step i. For **global variables** we define $R_{\text{globals}}\ eid\ g$ in terms of auxiliary functions reused from WasmCert. The **heap** relation $R_{\text{heap}}\ eid\ size$ is the most complicated. It relates the WasmCert memory type to the finite map returned by the mem_at function, and says that if you read 8 bytes from a word-aligned address in the abstract state, you get the same value as the word in the corresponding zkWasm MTable location. It also says that the memory size is tracked correctly, and that it is not possible to read data beyond the memory size. Finally, the **callstack** relation $R_{\text{callstack}}\ id\ f\ cs$ says that the JTable represents a linked list of control flow labels cs, starting in function f. Each list node has an identifier id, and the ETable column $\text{ETable}(\text{frame_id}, i)$ stores the id of the first node in the list.

Main theorem. One complication in stating our simulation theorem is ill-typed programs. For example, in our specification the Add rule (Figure 3) only applies if the stack contains at least two items. This reflects the official Wasm specification, which asserts properties that hold due to type checking of the Wasm program. In the case of the binary operation, the specification provides "Assert: due to validation, two values of value type are on the top of the stack." Since the Wasm program is type-checked as part of generating the verification key, users can rely on this assertion, but it is separate from the circuits, e.g. there is no explicit constraint to prevent a load from negative stack index, so there is nothing to stop an ill-typed Add from succeeding (with attacker-choosen inputs) if the stack is empty. We need to state our correctness theorem to exclude such cases.

To model this, we define a predicate $\text{Valid}(\sigma)$, by cases of the current instruction (Figure 7). It codifies the assertions in the Wasm specification, for example it says that if the current instruction is Bin then the stack has at least two items, if the current instruction is $\text{LocalSet}_s\ o$ then the stack has an item at depth o, etc. We define a predicate TypeSafe which states that every state reachable by executing the program (including the initial state) satisfies Valid:

$$\text{TypeSafe}(\sigma) := \forall \sigma'. \sigma \to^* \sigma' \implies \text{Valid}(\sigma').$$

Now we can state our soundness theorem. We prove that if the first row in the table represents the initial state of a Wasm program, then each of the subsequent rows represents a step in a correct Wasm execution. Since this is an upwards-simulation, we cover every possible malicious witness from an attacker.

However, the statement needs to be careful because a zkVM can accept private inputs. A first attempt looks similar to the correctness theorem of a normal compiler or interpreter:

Corollary 3.1 (soundness). *Suppose that* $\text{TypeSafe}(\sigma_{\text{initial}})$ *and* $R\ 0\ \sigma_{\text{initial}}$. *Then for all* i *such that* $0 \le i < \text{length}(\text{ETable})$ *and* $\text{ETable}(\text{enabled}, i) = 1$, *there exists a state* σ *such that* $\sigma_{\text{initial}} \to^* \sigma$ *and* $R\ i\ \sigma$.

We prove this as a corollary to our main Theorem 3.2, but in the zk setting it is not quite strong enough. The above statement says that there *exists* a Wasm execution, but intuitively, the person

$$\mathrm{R}_{\mathrm{stack}} : (eid : \mathbb{Z}) \to (sp : \mathbb{Z}) \to (s : \mathrm{list}\ \mathbb{Z}) \to \mathrm{Prop}$$

$$\frac{}{\mathrm{R}_{\mathrm{stack}}\ eid\ sp\ \mathrm{nil}} \qquad \frac{\mathrm{get}\ (\mathrm{memory_at}\ eid\ \mathrm{STACK})\ sp = \mathrm{Some}\ x \qquad \mathrm{R}_{\mathrm{stack}}\ eid\ (sp+1)\ xs}{\mathrm{R}_{\mathrm{stack}}\ eid\ sp\ (x :: xs)}$$

$$\mathrm{R}_{\mathrm{callstack}} : (id : \mathbb{Z}) \to (f : \mathbb{Z}) \to (cs : \mathrm{list}\ label) \to \mathrm{Prop}$$

$$\frac{}{\mathrm{R}_{\mathrm{callstack}}\ 0\ f\ \mathrm{nil}} \qquad \frac{(id, next_id, f, l) \in \mathrm{JTable} \qquad \mathrm{R}_{\mathrm{callstack}}\ next_id\ (\mathrm{fst}\ l)\ cs}{\mathrm{R}_{\mathrm{callstack}}\ id\ f\ (l :: cs)}$$

$$\begin{aligned}
\mathrm{R}_{\mathrm{globals}} &: (eid : \mathbb{Z}) \to (g : \mathrm{list\ Wasm.global}) \to \mathrm{Prop} \\
\mathrm{R}_{\mathrm{globals}}\ eid\ g &:= \forall jz.\ \mathrm{get}(\mathrm{memory_at}\ eid\ \mathrm{GLOBAL})\, j = \mathrm{Some}\ z \implies \mathrm{Wasm.glob_val}\ gs\ j = \mathrm{Some}\ z \\
\mathrm{R}_{\mathrm{heap}} &: (eid : \mathbb{Z}) \to (size : \mathbb{Z}) \to (mem : \mathrm{Wasm.memory}) \to \mathrm{Prop} \\
\mathrm{R}_{\mathrm{heap}}\ eid\ size\ mem &:= (\mathrm{Wasm.mem_length}\ mem)/\mathrm{page_size} = size \\
&\quad \wedge \ \forall block\ z\ .(8 * block + 8 \le \mathrm{mem_length}\ mem) \implies \\
&\qquad \mathrm{get}\ (\mathrm{memory_at}\ eid\ \mathrm{HEAP})\ block = \mathrm{Some}\ z \implies \\
&\qquad \exists bs.\ \mathrm{Wasm.read_bytes}\ mem\ (8 * block)\ 8 = \mathrm{Some}\ bs \wedge \mathrm{Wasm.decode_int}\ bs = z \\
&\quad \wedge \forall block\ z.\ \mathrm{get}\ (\mathrm{memory_at}\ eid\ \mathrm{HEAP})\ block = \mathrm{Some}\ z \implies (8 * block + 8) \le \mathrm{page_size} * size;
\end{aligned}$$

$$\begin{aligned}
\mathrm{R} &: (i : \mathbb{Z}) \to (\sigma : \mathrm{state}) \to \mathrm{Prop} \\
\mathrm{R}\ i\ \sigma &:= \sigma.\mathrm{pc} = (\mathrm{ETable}(\mathrm{fid}, i), \mathrm{ETable}(\mathrm{iid}, i)) \ \wedge\ R_{\mathrm{stack}}\ \mathrm{ETable}(\mathrm{eid}, i)\ (\mathrm{ETable}(\mathrm{sp}, i) + 1)\ \sigma.\mathrm{stack} \\
&\quad \wedge\ R_{\mathrm{globals}}\ \mathrm{ETable}(\mathrm{eid}, i)\ \sigma.\mathrm{globals} \\
&\quad \wedge\ R_{\mathrm{heap}}\ \mathrm{ETable}(\mathrm{eid}, i)\ \mathrm{ETable}(\mathrm{mpages}, i)\ \sigma.\mathrm{memory} \\
&\quad \wedge\ R_{\mathrm{callstack}}\ \mathrm{ETable}(\mathrm{frame_id}, i)\ \mathrm{ETable}(\mathrm{fid}, i)\ \sigma.\mathrm{callstack}
\end{aligned}$$

Figure 6: The simulation relation R

$$\begin{aligned}
&\mathrm{Valid}(\sigma) := \\
&\quad (program[\sigma.\mathrm{pc}] = \mathrm{BinBit_s}\ o \implies \exists x_r\ x_l\ xs.\ \sigma.\mathrm{stack} = x_r :: x_l :: xs) \\
&\wedge (program[\sigma.\mathrm{pc}] = \mathrm{Bin_s}\ o \implies \exists x_r\ x_l\ xs.\ \sigma.\mathrm{stack} = x_r :: x_l :: xs) \\
&\wedge \ldots
\end{aligned}$$

Figure 7: The Valid **predicate**

who prepared the witness should *know* that execution. The difference can be seen for example if the Wasm program checks for a collision of a cryptographic hash function: we expect the person who prepared the witness to actually know the input values that collided. In principle, a statement like Theorem 3.1 could have been proven by a non-constructive proof by contradiction, in which case it would only say that there exists an execution with a collision but not what that execution was. The standard way [3] to avoid this is to state *knowledge extraction*, saying that there exists a computable function from the witness to the Wasm execution.

We therefore define a function *extract* : nat → list State, computable in linear time, which extracts the first i states of the Wasm execution. It simply looks at each row of the witness table to calculate the next state. Writing $\sigma \xrightarrow{l}{}^{*} \sigma'$ to mean that the machine steps from σ to σ' via the intermediate states l, we have:

Theorem 3.2 (knowledge-soundness). *Under the same assumptions as above, there exists a state σ such that $\sigma_{\mathrm{initial}} \xrightarrow{extract(i)}{}^{*} \sigma$ and $R\ i\ \sigma$.*

4 The Proofs

Now we consider in more detail how Theorem 3.2 was proven. The proofs follow the structure of the system: we first prove the correctness of the auxiliary tables (MTable, JTable, and BitTable), and then use those lemmas for the main theorem about the ETable. It is split into 27 cases based on which Op bit is set.

4.1 Verifying the Memory Table

The most distinctive thing about a zkVM, compared to interpreters and hardware CPUs, is the way it represents the dynamically sized state (heap, value stack, call stack). ZkWasm uses a classic idea, originally by Ben-Sasson et al. [5, 6], to represent this by a sorted table of memory writes. In fact, it used two slightly different variations of the idea for the memory and for the call stack, differing by the precise counting scheme used. Prior work had not written down a detailed proof that it correctly represents the memory (instead focusing on how to efficiently zk-prove sortedness, how to parallelize the zk-proofs, etc). It seems intuitively obvious that it is correct, but to convince a proof assistant we need to break it down into a detailed sequence of lemmas. This was worthwhile, because in doing so we discovered that the zkWasm developers had made a mistake when designing the invariants of the call stack table (see Section 5.1).

Also, because we are developing a mechanized proof, the software engineering aspects are important. In a conventional bytecode interpreter the memory may be a dictionary mapping addresses to values, and in WasmCert the memory is modeled by a type `Wasm.memory` which is roughly an alias for `list byte`. So for a conventional interpreter one must prove the representation relation is preserved after one update in the dictionary and one update in the list. Our ambition is that the proof of each zkWasm instruction should not be much more complicated than this. The memory table invariants should be encapsulated in only the memory table part of the proof.

4.1.1 The contents of the memory table. Recall the example MTable in Figure 5. Each entry represents one memory write in the trace, together with the range of step numbers for which that write is valid.

The circuit for the Wasm memory Load instruction uses a lookup constraint to enforce that there exists an entry in the MTable such that the EID of the Load instruction is in the start–end range. But to directly write such constraints would be cumbersome. ZkWasm provides an abstraction layer in the form of two convenience functions

to allocate "lookup write" and "lookup read" cells. For example, a function call `memory_table_lookup_write_cell(eid, type, offset, value)` creates constraints to express "at instruction `eid`, `value` was written to the location `offset`" (`type` is stack, heap, or global). Then, when implementing the Wasm Store instruction, the circuit programmer can use these functions and think of the MTable as a conventional mutable memory.

4.1.2 Specifying the zkWasm memory abstraction. A verification development should have the same structure as the software being verified, so the proofs can follow the same logic as the code. Therefore, we want to verify the MTable circuits and the "alloc read/write cell" functions as a single module, with an interface that reflects the intention of representing a mutable memory. We do this by a Rocq function memory_at *eid type* which scans through the MTable and builds the corresponding key-value dictionary. We then prove that the constraints generated by the alloc functions are equivalent to mutating the corresponding dictionary using set and get functions. In simplified pseudo-code, we prove, for each *eid*:

- if the constraints `(memory_table_lookup_read_cell eid type offset value)` hold, then `get (memory_at eid type) offset = Some value`
- if the constraints `(memory_table_lookup_write_cell eid type offset value)` hold then `(memory_at (eid+1) type) = set (memory_at eid type) offset value`.

4.1.3 The zkWasm counting scheme. We have not yet explained the full contents of the MTable and JTable. The attacker controls the tables, so with the format described so far, an attacker could simply insert an extra row to make a memory load instruction return an arbitrary value.

To ensure that the MTable and JTable only contain entries generated by actual return and call instructions, zkWasm uses a scheme to count the number of entries. Specifically, the MTable has a column that keeps a running count of the number of entries, while the execution table has a column mops that counts the intended number of memory write operations of each instruction. One equality constraint ensures that the counts are equal, i.e., $\mathsf{ETable}(\mathsf{mops}, 0) = \mathsf{MTable}(\mathsf{mops}, 0)$. The intuition is simple: each memory write in the ETable is counted, and there are exactly that many entries in the MTable, so there can not be any extra entries created by an attacker.

That intuitive statement is too hand-wavy to be accepted by a mechanized proof assistant, so we need to make it precise. As a first step, we need to correct the statement of the writing lemma above. We define a function `mops_at eid type` which counts the number of MTable entries with a given eid and type (most instructions will create 0 or 1 entries at one eid). The correct statement of the theorem has one more precondition, saying that there were no spurious MTable entries:

- if the constraints `(memory_table_lookup_write_cell eid type offset value)` hold and **`(mops_at eid type) = 1`** then `(memory_at (eid+1) type) = set (memory_at eid type) offset value`.

We can already see that this forces us to be more precise than the intuition above. It is not enough to prove that the total number of rows in MTable is equal to the total number of memory writes in the trace (which would follow easily from the equality constraint). In order to prove the correctness of the instructions we need to know that *each instruction* corresponds to the right number of MTable entries. For example, we need to rule out that the attacker omitted the entry for one instruction in the trace and created a malicious extra entry for a different instruction.

The counting lemmas. To prove that each instruction creates the right number of entries, we use an argument to bound the number of entries from above and from below, which works in three steps. First, for each type of instruction we define how many memory entries it should create, and we call the intended number of writes of ith instruction in the trace `i_mops i`.

Then, consider the tail of the trace from the ith instruction onwards. We call the intended number of writes from the ith element to the end of the trace **`i_mops i`**, and we call the actual number of entries in the MTable from the ith instruction to the end $\mathsf{cum_mops}(\mathsf{ETable}(eid, i))$. Each segment $i \ldots \mathrm{length}(\mathsf{ETable})$ of the trace creates at least as many entries as intended:

Lemma 4.1. $\mathsf{cum_mops}(\mathsf{ETable}(\mathsf{eid}, i)) \geq \mathsf{is_mops}\ i$.

Proof. By analyzing the lookup-constraints for each instruction (case by case), we prove that the instruction creates at least as many entries as intended. Then just add up all the instructions in the tail of the trace. □

From this it follows trivially that if the table also has at most as many entries as intended, then it has exactly the right number:

Lemma 4.2. *If* $\mathsf{cum_mops}(\mathsf{ETable}(\mathsf{eid}, i)) \leq \mathsf{is_mops}\ i$, *then* $\mathsf{cum_mops}(\mathsf{ETable}(\mathsf{eid}, i)) = \mathsf{is_mops}\ i$.

Now we are ready for step three. We want to prove that for every n, cum_mops and is_mops agree for the part of the tables from n to the end:

Lemma 4.3. *For all* $0 \leq n < length(\mathsf{ETable})$, $\mathsf{cum_mops}(\mathsf{ETable}(\mathsf{eid}, n)) = \mathsf{is_mops}\ n$.

Proof. By induction on n. For the very first row in the table, we have the zkWasm equality constraint saying that the total number of entries in MTable is correct, i.e. $\mathsf{cum_mops}\ 0 = \mathsf{is_mops}\ 0$.

For a later row, the induction hypothesis tells us that $\mathsf{cum_mops}\ n = \mathsf{is_mops}\ n$ and we want to prove $\mathsf{cum_mops}\ (n+1) = \mathsf{is_mops}\ (n+1)$. Note that $\mathsf{cum_mops}\ n = \mathsf{mops_at}\ n + \mathsf{cum_mops}\ (n+1)$ and $\mathsf{is_mops}\ n = \mathsf{i_mops}\ n + \mathsf{is_mops}\ (n+1)$. So we have $\mathsf{mops_at}\ n + \mathsf{cum_mops}\ (n+1) = \mathsf{i_mops}\ n + \mathsf{is_mops}\ (n+1)$. We previously proved that each instruction creates at least as many entries as it should, i.e. that $\mathsf{mops_at}\ n \geq \mathsf{i_mops}\ n$. So it follows that $\mathsf{cum_mops}\ (n+1) \leq \mathsf{is_mops}\ (n+1)$ which is exactly what we need to apply Lemma 4.2. □

4.1.4 Making the Proof Modular. From the discussion above, we can see that there is a circular dependency between the proofs about per-instruction circuits and the proofs about the count column of the ETable. To prove that the instruction circuits are correct, we need to reason about memory writes in them; which requires knowing the number of MTable entries at that eid; which requires proving the memory write count in the execution table is correct;

which requires knowing that each instruction makes at least a minimum number of memory writes.

To structure the verification so that the work can be split up between several engineers, we divide each instruction correctness theorem into two steps. For each instruction type, zkWasm specifies an expected number o of memory operations. Taking the Xor instruction as our example again, the verification engineer proves both a simpler theorem `opcode_mops_correct_bin_bit` which says that the instruction creates *at least* o MTable entries, and also the full correctness theorem `BitOp_Xor_correct`, which takes an hypothesis that the instruction creates *exactly* o MTable entries. These two theorems can be proven independently. The proof of the final theorem uses the mops-correct lemma and Lemma 4.3 to satisfy the "exactly o" precondition. Notably, all the per-instruction proofs can be done at the level of read/write constraints, without needing to know anything about the implementation of the MTable. The project therefore divides into three independent parts: the memory system, the per-instruction proofs, and the top-level proof about the ETable.

4.2 Verifying the JTable and BitTable

In addition to the MTable, zkWasm has two other important auxiliary tables.

The JTable is used to store the return addresses for the call stack. It uses a counting scheme very similar to that of the MTable (after fixing the bug in Section 5.1), and the proofs for that are essentially the same as what we described above (with some extra work to deal with idiosyncratic table encoding).

The BitTable is used to compute the values for the bitwise operations (AND, OR, XOR, and PopCount). For each 64-bit operation that occurs in the program trace, it contains 11 adjacent rows. Eight of them contain the bytes from the input words, and the result of applying the bitwise operation to those bytes, and these are accumulated by gate constraints into a row with the full word. The final correctness statement for the BitTable says that the row at the beginning of each block contains a correct tuple of numbers $(l, r, l \text{ OP } r)$. Verifying it only requires noting that the bitwise operations distribute over additions and bitshifts.

4.3 Verifying the Instruction code

With all the lemmas about auxiliary tables complete, it is finally possible to verify the correctness theorems for all the individual Wasm instructions. These proofs follow a common pattern. First, we must prove a "decode" lemma, saying that, e.g., if OpBinBit = 1 and op = XOR, then the current pc points at the encoding of an BinBit_s XOR instruction in the ImTable. This requires proving that the encoding function is injective. Then we use lemmas about stack handling to prove that some particular table columns contain the top items on the stacks, then use the instruction-specific circuits to prove that the content of some other table column has the right result, and finally prove that the stack and memory at the next row contain the new values.

The different instructions vary in complexity. One can get a sense of what parts of a zkVM are easy or difficult by considering the size of the proof devoted to them (Figure 8). The lemmas related to the stack accesses are in RelationHelper and are shared for all the

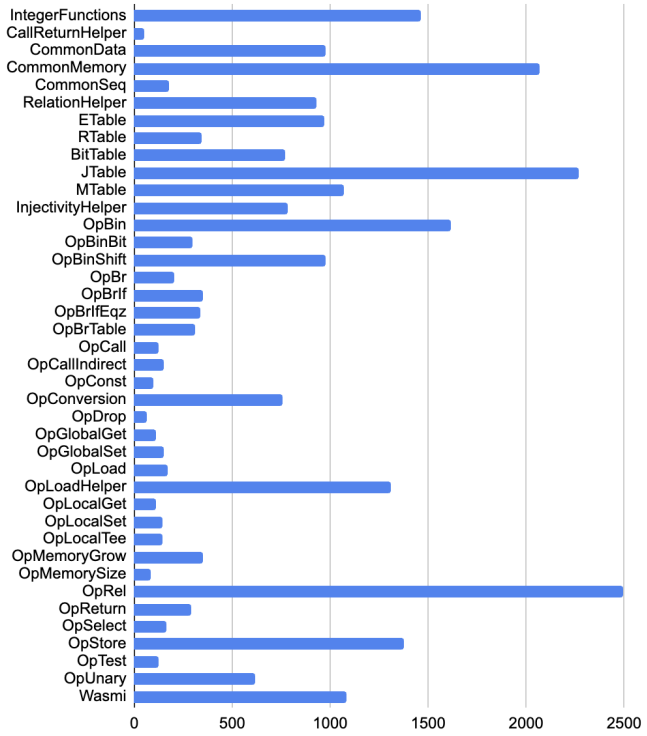


Figure 8: Lines of proofs for the main components of the development

instructions. The 27 different types of Wasm instructions are named OpXX. Some are trivial, e.g., the implementation of the MemorySize instruction simply copies one of the ETable columns (which tracks maximum accessible address) to the stack.

One outlier is IntegerFunctions, which proves three theorems characterizing the "count leading/trailing zeros" and "count bits" functions (e.g., if $x = z2^{y+1} + 2^y$ then x has y trailing zeros), as well as theorems about bitwise and/or/xor. Here we pay a price for reusing the WasmCert definitions of these bitwise functions, which were not written with our verification in mind and made these simple statements surprisingly hard.

InjectivityHelper proves correctness of packing multiple integers together by bit-shifting and adding. Here we prove the operation $(x, y) \mapsto (x \ll s) + y$ is injective if $0 \le y < 2^s$, and use custom ltacs to recursively apply this to the complete packed word.

The most complex instructions to implement are the memory load and store instructions (CommonMemory + OpLoadHelper + OpLoad + OpStore = 4935 lines of proof). This reflects the large amounts of zkWasm circuit code (1200 lines). The reason is that zkWasm needs to bridge the gap between the byte-based memory specified by the Wasm language, and the memory based on 64-bit words that the MTable implements. Wasm memory Store instructions are not required to be word-aligned, so to implement the instruction zkWasm needs to access two words in the MTable, update the relevant bytes in them, and write them back. Expressing this as zk-constraints requires many equations.

The next most complex instructions are the arithmetic operations on numbers (OpBin). This is a contrast to conventional bytecode interpreters, where the instructions to add, subtract, divide, and multiply are trivial. The difference here is that zkWasm needs to handle integer overflow carefully, since the native arithmetic is modulo p, so the input numbers must be decomposed into 16-bit words with equations between them. One can see that on a line-by-line level, the greatest source of complexity in the zkVM is due to the mismatch between bit-based numbers and the zk-prover's native field arithmetic.

5 Evaluation

5.1 Bugs found

Missing constraint for LOAD8. Wasm has instructions Load64, Load32, Load16 and Load8 to read variously-sized words from the heap. The implementation is complicated (about 600 lines of constraints) because the Wasm memory is addressed by byte while the zkWasm MTable contains 64-bit words. One of the Load gates was meant to force the high bits of the loaded value to be zero if loading less than 64 bits:

```
vec![ is_four_bytes.expr(m) *
 (load_picked.u16[2].expr(m) + load_picked.u16[3].expr(m)),
is_two_bytes.expr(m) *
 (load_picked.expr(m) - load_picked_leading_u16.expr(m)),
is_one_byte.expr(m) *
 (load_picked_leading_u16_u8_high.expr(m))]
```

Unfortunately the last line is wrong, and should actually be

```
is_one_byte.expr(m) *
 (load_picked.expr(m) - load_picked_leading_u16_u8_low.expr(m))
```

With the incorrect version, the only constraint on `load_picked` when using a LOAD8 instruction was that 8 bits in the middle of a 64-bit value had to be 0. An attacker could target any program that contains a LOAD8 instruction and cause any such load operation in the trace to return a value larger than 8 bits. Because memory loads are a basic primitive of program execution, it is quite likely that this is enough to create a critical fault in the program.

Incorrect instruction decoding for Conversion. The Wasm conversion instruction casts between different-sized integer types. When converting from a 64-bit to a 32-bit number it is supposed to truncate the leading bits, and the Etable has a bit column which controls if the truncation constraint is active. The intention was that if the conversion operation is Conv $64to32$, then the wrap bit should be set, but the circuits accidentally stated the opposite implication, that if the bit is set then the conversion was from 64 to 32.

Therefore the bit is underconstrained. An attacker can create a trace where it is not set, and a too big number is put on the stack. It is possible that this would not be exploitable, because reads *from* the stack in zkWasm carry a 32/64 bit typecheck, but in a slightly different setting this could also have been a critical error.

These bugs are not "deep," but they show how tricky zkVM programming can be. In a conventional bytecode interpreter, failing to load a value would make all test cases fail, but zk-circuits will still accept all the intended traces, so the error is not immediately apparent.

Incorrect counting scheme for JTable. The third bug is even more subtle, and had to do with the counting scheme for the JTable. The JTable represents the call return stack as a linked list and contains one entry for each CALL instruction in the program. As with the MTable, there needs to be a count to ensure that there are no "extra" rows in the JTable. This was very similar to the scheme for the MTable, but with the difference that both call and return instructions in the ETable contributed 1 to the count, and each entry in the JTable carried a count of 2. Figure 9a illustrates how this works for a valid trace. The intended behavior is that every entry in the JTable will be referenced by exactly one call and one return instruction; the intuition is the same as for MTable: every instruction in the ETable is counted, so there is no space in for the attacker to insert extra rows into the JTable.

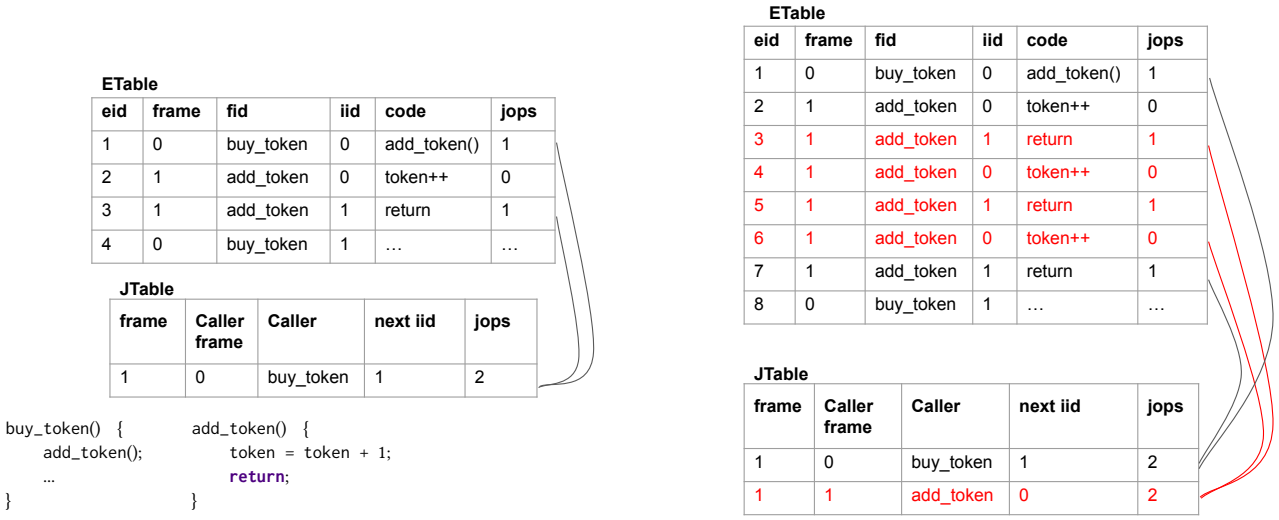
However, although intuitively the design seems okay, the more detailed lemmas in Section 4.1.3 do not go through. And in fact, there is an issue. While a call plus a return results in a jops count of 2, so do two calls or two returns. A valid trace should not have two calls or returns per frame, but this is precisely what attackers might exploit. An attacker can inject faked returns (and corresponding frames) into the ETable and JTable. For example, the previous ETable and JTable example of `buy_token` calling `add_token` can be attacked as in Figure 9b, by adding two faked returns between the original call and return in the ETable and a new faked frame row in the JTable (the execution steps for the original return and subsequent instructions must be incremented by 4 in the ETable). Since the jops count of each row in the JTable is 2, the counting constraints are satisfied. The attacker has the freedom to make the control flow jump to an almost arbitrary location; in this example the `token` variable is incremented three times instead of once.

This bug is a good example of the value of formal verification. It was not found through simple code review, and indeed there is no single line of code that is wrong. It seems very difficult to find through fuzzing or other testing, because it needs a very particular execution trace. And the table design seems plausible until one writes down the properties in mathematical detail.

5.2 Effort

All of zkWasm contains around 22,000 lines of code, but for soundness we only need to target the `circuits` subfolder of zkWasm. It contains around 6,000 lines of Rust, but not all of that is constraint code, because the files also have some functions to produce the witness. The development contains 39,936 lines of Rocq (14,030 definitional, 25,906 proof) as counted by `coqwc`. The definitions are mostly the translated circuits (Section 3.1).

The total work for the verification was 53 person-weeks, of which 22 pw was spent on the auxiliary tables (BitTable, MTable, JTable, and shared columns of the ETable) and 31 pw was spent on verifying each individual instruction. The time spent roughly corresponded to the line counts in Figure 8. The team consisted of two verification experts plus two engineers with no prior formal verification experience. First a single person proved the BitTable and MTable correct and defined the refinement relation, and then additional engineers worked in parallel to verify the instructions. Ultimately, verification is economically feasible: the total person-weeks is much smaller than the effort to write the zkVM itself.



ETable

eid	frame	fid	iid	code	jops
1	0	buy_token	0	add_token()	1
2	1	add_token	0	token++	0
3	1	add_token	1	return	1
4	0	buy_token	1	…	…

JTable

frame	Caller frame	Caller	next iid	jops
1	0	buy_token	1	2

```
buy_token() {
    add_token();
    …
}
```

```
add_token() {
    token = token + 1;
    return;
}
```

(a) The intended tables for the example program

ETable

eid	frame	fid	iid	code	jops
1	0	buy_token	0	add_token()	1
2	1	add_token	0	token++	0
3	1	add_token	1	return	1
4	1	add_token	0	token++	0
5	1	add_token	1	return	1
6	1	add_token	0	token++	0
7	1	add_token	1	return	1
8	0	buy_token	1	…	…

JTable

frame	Caller frame	Caller	next iid	jops
1	0	buy_token	1	2
1	1	add_token	0	2

(b) Tables constructed by an attacker

Figure 9: Attacking the incorrect JTable counting scheme

5.3 Reducing effort through automation

The proofs are longer than they essentially have to be due to the tool we used. We chose Rocq because it had a well-tested specification of Wasm available. But Rocq's proof automation is comparatively weak for arithmetic goals. As an experiment, we ported one of the proof files, BitTable, into the Verus prover.

We use the original proof as a blueprint, so the ported development follows exactly the same structure, with the same axioms and the same lemma statements. But because Verus is based on an SMT-solver the proofs get shorter: the original file had 727 lines of tactics and rewrite hints, while the ported file had 292 lines of proof function bodies (40%). This count somewhat underestimates the difference in effort, because Rocq tactics need more attention while many of the lines of Verus just mention the needed lemma and lets the SMT solver figure out how to use it.

Further, we no longer needed 1033 lines of proofs about properties of bitwise functions (the part of IntegerFunctions used by BitTable), because Verus's bitvector theory can prove them automatically. So the total amount of proof for the bit table decreased from 1760 to 292 lines (17%), but other zkWasm modules would not see as dramatic improvement.

The main reason we get this improvement is that the WasmCert semantics, and hence our Wasmi semantics, defines all the bitwise operations using the Rocq $\mathbb{Z}$ integer type. The Rocq standard library has a few lemmas about the bit representation of integers, but it is very rudimentary, and we had to prove even the most basic properties about bitwise-and and -or ourselves. This is very different from working in Verus, where the u64 type is directly backed by an SMT solver with support for bitvector theories. There are also efforts to integrate SMT solvers with Rocq, e.g. SMTCoq [1], so it may be that some of the same automation benefits could also be realized inside an interactive proof assistant.

6 Related and Future Work

Verification of zkVMs is an active topic. Apart from zkWasm, there are multiple projects which all target the RISC-V instruction set: OpenVM, Risc Zero, SP1, Nexus zkVM, Jolt. Two of them have verification efforts, which were done concurrently with our work (and announced after us).

Kwan et al. [16] verified the Jolt instruction constraints using ACL2. They do not prove the memory, only the register-to-register part of the instructions, e.g. their LOAD instruction assumes there is some loaded value x, and only proves that it is correctly bit-extended. Thus, the work we do to prove the correctness of the memory representation (Section 4.1), and to structure the proofs to abstract the memory away, have no analog there. The instruction proofs are quite different from ours because the Jolt constraints are not PLONKish, and instead uses a novel form of "pure lookup argument" [22]. The Jolt creators believe that this makes the constraint system simpler (and presumably easier to prove correct), but so far other zkVMs use arithmetic constraints like zkWasm, so it is interesting to know that both kinds of constraints can be verified.

Nethermind [18] verified OpenVM using Lean. Similarly to Jolt, the proofs of the instructions are not connected to the memory subsystem, the theorem statement assumes that the loaded value is equal to the memory. They separately prove a statement about ordering values in the memory bus, and say that it "means" that that memory reads agree with previous writes. If they later want to prove an end-to-end theorem, we expect our approach to specifying the memory subsystem will be useful.

In this paper, we manually translated the circuits from Halo2 to Rocq, but the above projects automatically extract them. OpenVM still manually writes human-readable circuits in Lean, but then proves that those are equivalent to an AST pretty-printed by the zk-backend. A similar approach would be possible for zkWasm also.

Neither of these works have a single simulation theorem like ours (or a knowledge extractor). They prove separate correctness

theorems for each of the instructions, each of which combines operational semantics and refinement relation for that instruction.

Prior to these, the only verified zkVM was the Cairo virtual machine [2] in Lean. However, the Cairo VM is designed to directly correspond to zk-primitives, and is simplistic compared to a general-purpose language like Wasm or Risc-V. First, Cairo does not have mutable memory, only three registers and a ROM, so there is no need to verify a memory abstraction. The ROM is implemented by a simple table of pairs (address, value) with monotonically increasing addresses, so there is no need for the complicated table invariants in Section 4.1. Second, the registers contain field elements modulo p and there are only arithmetic operations, so there is no need to handle integer overflow or to implement bitwise operations. The Cairo computational model is interesting for providing zk-capabilities in a lightweight way, but it cannot serve as a target for conventional programming languages.

Before zkVMs, the state of the art was to compile entire programs into giant constraint sets, and some work aim to verify such transformations [15, 19, 21].

Future work. We hope to increase the efficiency of the proof process. The current work can serve as a "blueprint" which shows how the correctness of a PLONKish zkVM can be broken down into lemmas, but we believe that by using an SMT-based prover as in Section 5.3, the individual proofs could be made shorter. Also we would like to use the macro system in Verus to automatically generate the axioms directly from the Rust source code.

7 Conclusion

Zero-knowledge virtual machines are an exciting new technology, but to achieve acceptable security levels formal verification will be crucial. The implementation is quite different from a conventional interpreter, but we can adapt programming language methods to this setting to ensure it is sound.

Acknowledgments

This work was supported in part by Ethereum Foundation grants FY25-1977 and FY25-1978, an NSF CAREER Award CCF-2239484, an Amazon Research Award, a VMware Systems Research Award, and an NSF grant CCF-2124080. Ronghui Gu and Zhong Shao are co-founders of and have an equity interest in CertiK.

References

[1] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. A modular integration of SAT/SMT solvers to coq through proof witnesses. In *Proceedings of the First International Conference on Certified Programs and Proofs* (Kenting, Taiwan) *(CPP'11)*. Springer-Verlag, Berlin, Heidelberg, 135–150. doi:10.1007/978-3-642-25379-9_12

[2] Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman. 2022. A verified algebraic representation of cairo program execution. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 153–165. doi:10.1145/3497775.3503675

[3] Mihir Bellare and Oded Goldreich. 1993. On Defining Proofs of Knowledge. In *Advances in Cryptology — CRYPTO' 92*, Ernest F. Brickell (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 390–420.

[4] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. 459–474. doi:10.1109/SP.2014.36

[5] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. 2013. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: extended abstract. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science* (Berkeley, California, USA) *(ITCS '13)*. Association for Computing Machinery, New York, NY, USA, 401–414. doi:10.1145/2422436.2422481

[6] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *Advances in Cryptology – CRYPTO 2013*, Ran Canetti and Juan A. Garay (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 90–108.

[7] Vitalik Buterin. 2022. The different types of ZK-EVMs. https://vitalik.eth.limo/general/2022/08/04/zkevm.html.

[8] Vitalik Buterin. 2024. Possible futures of the Ethereum protocol, part 4: The Verge. https://vitalik.eth.limo/general/2024/10/23/futures4.html.

[9] Tornado Cash. 2019. Tornado.cash got hacked. By us. https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8.

[10] Chainlink. 2023. Zero-Knowledge Proof: Applications and Use Cases. https://chain.link/education-hub/zero-knowledge-proof-use-cases.

[11] Guillaume Claret and contributors. 2024. rocq-of-rust: Formal Verification Tool for Rust. https://github.com/formal-land/rocq-of-rust. Accessed: 2026-07-31.

[12] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive, Paper 2019/953. https://eprint.iacr.org/2019/953

[13] Sinka Gao, Guoqiang Li, and Hongfei Fu. 2024. ZKWASM: A ZKSNARK WASM Emulator. *IEEE Transactions on Services Computing* (2024), 1–14. doi:10.1109/TSC.2024.3422798

[14] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (Aug. 2022), 31 pages. doi:10.1145/3547647

[15] Kunming Jiang, Devora Chait-Roth, Zachary DeStefano, Michael Walfish, and Thomas Wies. 2023. Less is more: refinement proofs for probabilistic proofs. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1112–1129. doi:10.1109/SP46215.2023.10179393

[16] Carl Kwan, Quang Dao, and Justin Thaler. 2024. Verifying Jolt zkVM Lookup Semantics. *IACR Cryptol. ePrint Arch.* (2024), 1841. https://eprint.iacr.org/2024/1841

[17] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (Dec. 2009), 363–446. doi:10.1007/s10817-009-9155-4

[18] Nethermind. 2026. Formal verification of the OpenVM RISC-V zkVM. https://github.com/openvm-org/openvm-fv

[19] Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Clark Barrett. 2023. Bounded Verification for Finite-Field-Blasting. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 154–175.

[20] Andrew Shaughnessy. 2024. Advancing the State of the Art in Zero Knowledge Proofs: SIEVE Wrap Up. https://galois.com/blog/2024/09/advancing-the-state-of-the-art-in-zero-knowledge-proofs-sieve-wrap-up/.

[21] Jon Stephens, Shankara Pailoor, and Isil Dillig. 2025. Automated Verification of Consistency in Zero-Knowledge Proof Circuits. In *Computer Aided Verification: 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part I* (Zagreb, Croatia). Springer-Verlag, Berlin, Heidelberg, 315–338. doi:10.1007/978-3-031-98668-0_16

[22] Justin Thaler. 2023. A technical FAQ on Lasso, Jolt, and recent advancements in SNARK design. https://a16zcrypto.com/posts/article/a-technical-faq-on-lasso-jolt-and-recent-advancements-in-snark-design/.

[23] Morgan Thomas. 2022. Arithmetization of $\Sigma^1{}_1$ relations in Halo 2. *IACR Cryptol. ePrint Arch.* (2022), 777. https://eprint.iacr.org/2022/777

[24] Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. 2021. Two Mechanisations of WebAssembly 1.0. In *Formal Methods*, Marieke Huisman, Corina Păsăreanu, and Naijun Zhan (Eds.). Springer International Publishing, Cham, 61–79.

[25] Zcash. 2024. The Halo2 zero-knowledge proving system. https://github.com/zcash/halo2.

A Open Science

The source code of zkWasm itself was already publicly available before our work. In our artifact we supply the complete Rocq proofs, as well as the Verus experiment mentioned in Section 5.3.

B Ethical Considerations

The work was done in collaboration with the zkWasm team. All vulnerabilities were responsibly disclosed to the team as soon as they were discovered.