CERTIK

Formal Verification of zkWasm

Vilhelm Sjöberg
EF Workshop: Formal Verification of ZKP
October 21, Zürich

Verifying zkWasm

zkWasm is a full zkVM for WebAssembly (except the floating-point instructions).
User writes an arbitrary Wasm program, sets up an application key.
zkWasm prover runs the program with some inputs i and creates a proof object.
zkWasm verifier takes key and proof object, and accepts only if the program accepted the input i.

22k lines of Rust, ~6k is circuit constraints.

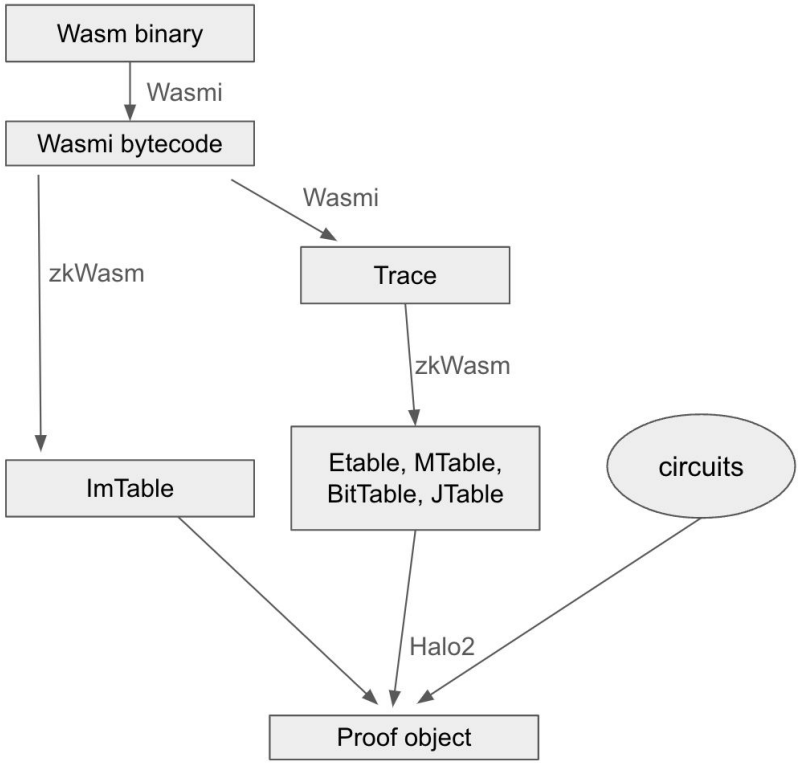
CertiK verified the circuits using Coq (33k lines).

We prove *soundness*: any valid witness represents a valid execution of the Rust program.

Structure of zkWasm

ZkWasm is written in Rust, and relies on the **Wasmi** interpreter to execute wasm programs, and the **Halo2** zk-prover to generate proof objects.

- Setup: compile the program and generate image table
- Prove: Run the program, translate trace into tables, use Halo2 to generate proof object



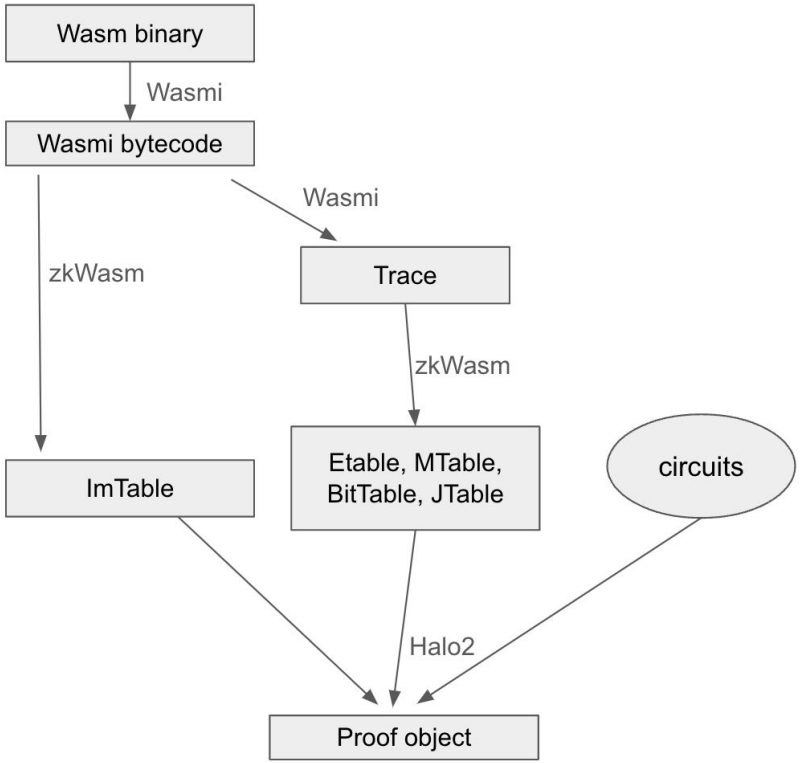
Structure of zkWasm

No no-op instructions. Implicit returns via reaching function scope end are replaced with an explicit return instruction.

Control flow structures (**while, if, case**) are flattened to **goto**s with a program counter.

The value stack is represented as a stack pointer. Locals live on the value stack. (There separate memory spaces for stack, heap, locals, and also instruction image.)

Halo2 to generate proof object



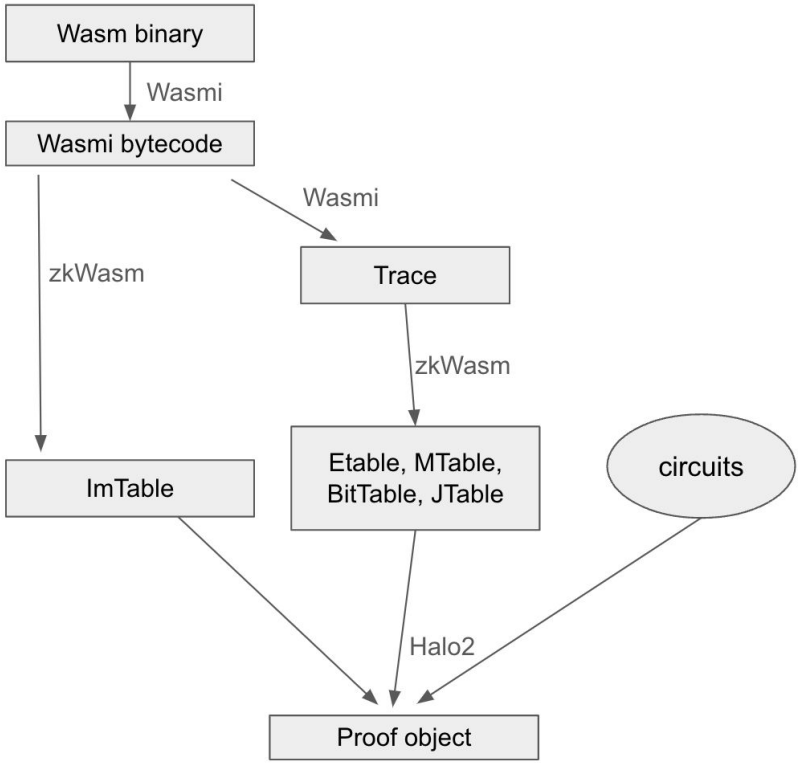
Structure of zkWasm

ZkWasm is written in Rust, and relies on the **Wasmi** interpreter to execute wasm programs, and the **Halo2** zk-prover to generate proof objects.

- Setup: compile the program
-

Contains program text and initial values for global variables.

This is a “fixed” table in Halo2: in the setup phase it’s used to create a verification key.



Structure of zkWasm

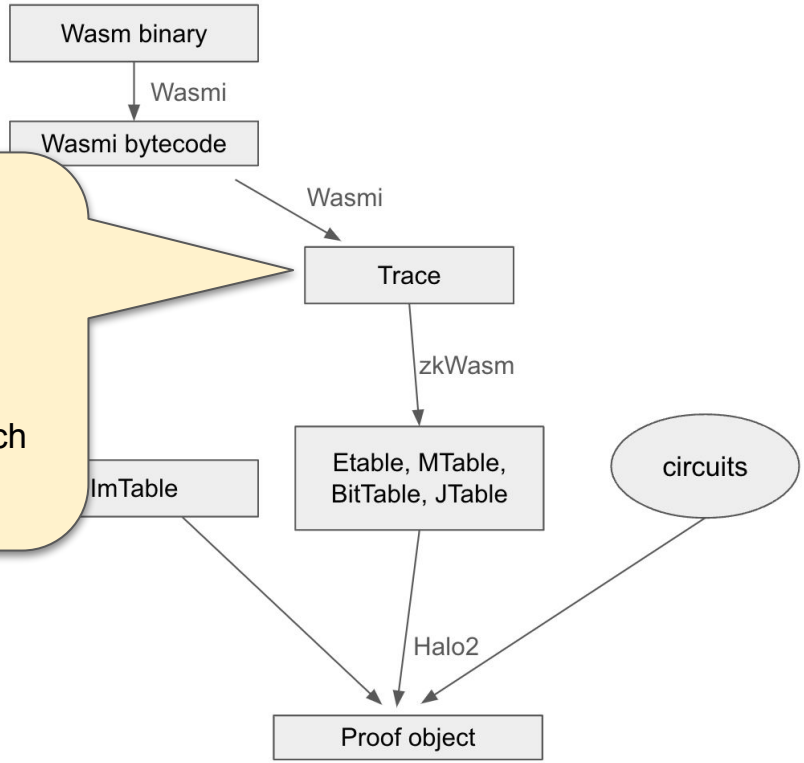
ZkWasm is written in Rust, and relies on the W… wasm… zk-pro…

- …
- …

Halo2 to generate proof object

The program should contain a function called `zkmain`, which can use the library functions `wasm_input` to read a sequence of integers and `require` to throw an error if some condition is not satisfied.

zkWasm hooks into Wasmi to record a trace of each instruction executed.



Structure of zkWasm

ZkWasm
on the **W**
wasm pro
zk-prover

- Set
  and
- Pro
  tran
  Hal

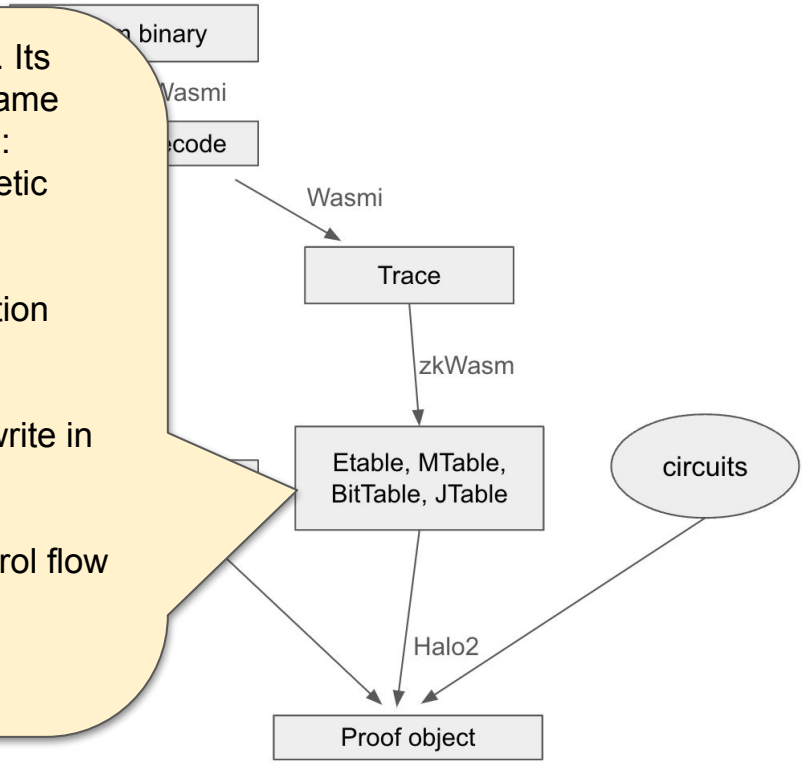
ETable has one row for each step in the trace. Its columns contain fixed-size state (mostly the same thing that would go into the registers of a CPU: program counter, inputs and outputs of arithmetic instructions, …)

BitTable contains data for each bitwise operation executed in the trace, rows for each byte.

MTable has rows for each heap/stack/global write in the trace.

JTable has rows for each address on the control flow return stack.

RangeTable enumerates integers $0…2^k$



Structure of zkWasm

ZkWasm is written in Rust, and relies on the **Wasmi** interpreter to execute wasm programs, and the **Halo2** zk-prover to generate proof objects.

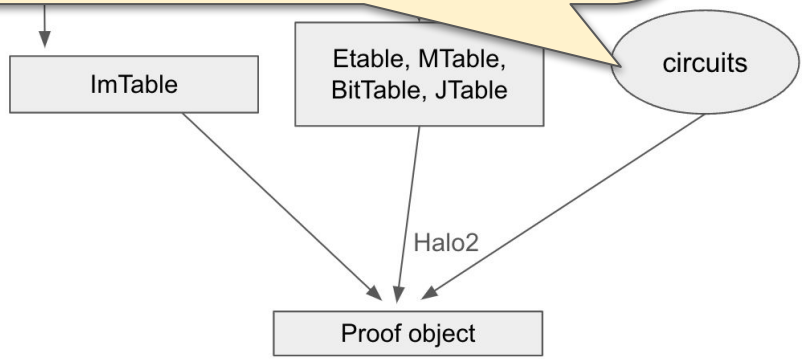
- Setup: compile the program and generate image table
- Prove: Run the program, translate trace into tables, use Halo2 to generate proof object

The part we are verifying!

The **circuits** directory in zkWasm is 12k lines of Rust (about 6k actual circuit definitions), out of 22k lines for the full zkWasm project.

(Only the circuits matter for the soundness result, the trace generation/witness construction does not need to be trusted.)

“PLONKish” **Fixed**, **Equality**, **Gate** and **Lookup** constraints.



Structure of zkWasm: ETable constraints

Each table has a set of circuits. Most are for the ETable, (27 files for the different Wasmi instruction classes).

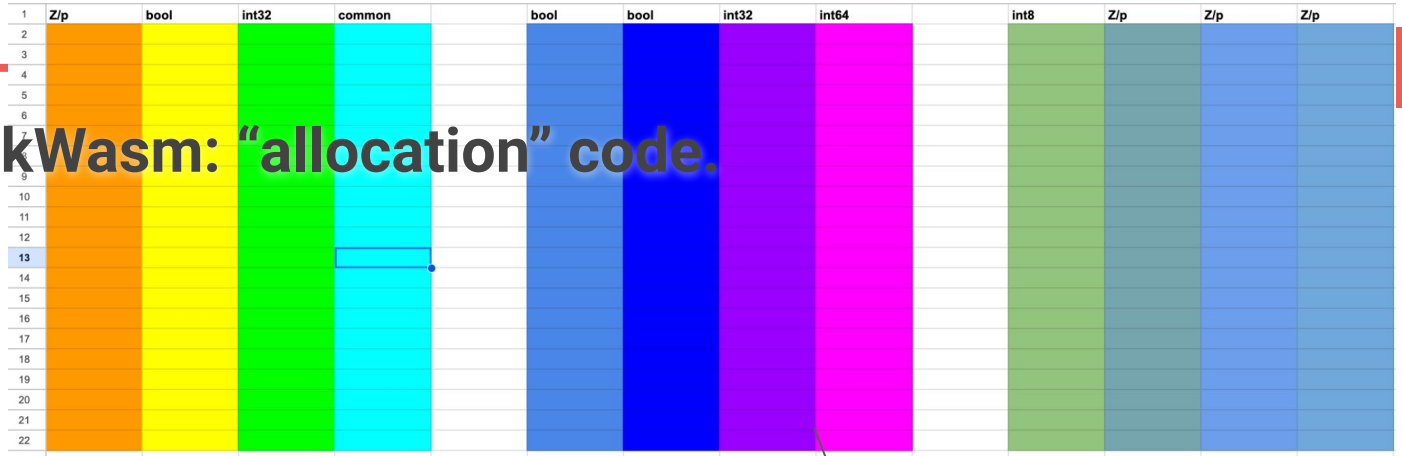
Each instruction has an ‘enable’ column in the ETable (0 or 1), and several data columns. The per-instruction constrains are multiplied by the enable bit.

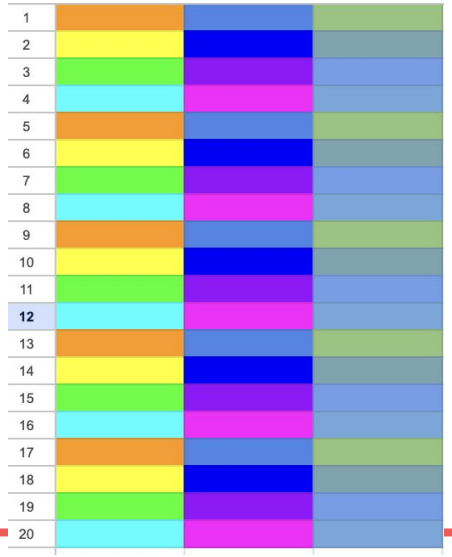
Each instruction then has many Halo2 arithmetic constraints, e.g.

```
constraint_builder.push(
    "op_load cross_block",
    Box::new(move |meta| {
        vec![
            is_cross_block.expr(meta) * constant_from!(WASM_BLOCK_BYTE_SIZE)
                + cross_block_rem.expr(meta)
                - load_inner_pos.expr(meta)
                - len.expr(meta)
                + constant_from!(1),
            cross_block_rem.expr(meta) + cross_block_rem_diff.expr(meta)
                - constant_from!(WASM_BLOCK_BYTE_SIZE - 1),
            (is_cross_block.expr(meta) - constant_from!(1))
                * load_value_in_heap2.expr(meta),
        ]
    }),
```

rtable	✅
image_table	✅
mtable	✅
jtable	✅
etable/op_bin	✅
etable/op_bin_bit	✅
etable/op_bin_shift	✅
etable/op_const	✅
etable/op_conversion	✅
etable/op_drop	✅
etable/op_global_get	✅
etable/op_global_set	✅
etable/op_local_get	✅
etable/op_local_set	✅
etable/op_local_tee	✅
etable/op_load	✅

Structure of zkWasm: “allocation” code.





The circuits are written with respect to multiple “logical” tables with typed columns.

Some Rust code code merges this into a single “physical” Halo2 table with interleaved columns + constraints to enforce the types.

There is about 1000 lines of Rust code to implement this. We consider it as trusted, and verify the circuits at the logical level.

Machine Model in Coq: Tables

We translate the circuit definitions from the Halo2 Rust eDSL intro Coq. We did this by translating the zkWasm code by hand. (Future work could change this.)

Our approach is a shallow embedding into Coq. Each of the Halo2 tables are postulated as **Parameters**, and all the circuits are turned into **axioms** about the tables.

11k lines of Coq, with comments specifying what Rust file they correspond to.

```
Inductive bit_table_cols :=
  | block_sel
  | u32_sel
  | lookup_sel
  | op
  | helper
  | val_l
  | val_r
  | val_res.

Parameter bit_table_values : bit_table_cols -> Z -> Z.
Parameter bit_table_numRows : Z.
```

Machine Model in Coq: Gates

The Rust source is a combination of calls to Halo2 “builders” and Rust macros. We ignore the phase distinction and translate both to Coq functions.

We were able to translate the constraints almost exactly line by line; happily the style of zkWasm is a good match for our style of Coq model. (Exception: one macro to multiply ETable constraints with the “enable” bits, 609 lines out of 6000.)

```
macro_rules! acc_u32_if_popcnt {
    ($col:ident) => {
        fixed_curr!(meta, u32_sel)
            * (acc_u32_helper!($col) - curr!(meta, $col))
            * is_popcnt
    };
}


vec![
    compose_u32!(left),
    compose_u32!(right),
    compose_u32_if_bit!(result),
    acc_u32_if_popcnt!(result),
]
```

```
Definition acc_u32_if_popcnt col :=
    (get u32_sel 0)
    * (acc_u32_helper col - (get col 0))
    * is_popcnt.

Axiom bit_table_gate_2 :
  gate bit_table
    (fun get =>
      compose_u32 get val_l ::
      compose_u32 get val_r ::
      compose_u32_if_bit get val_res ::
      acc_u32_if_popcnt get val_res :: nil).
```

Machine Model in Coq: Allocation & Range Axioms

Because the allocator code is trusted, we have separate column names, and we introduce axioms expressing the column types (so we trust that the allocator got those constraints right).

```
let enabled_cell = allocator.alloc_bit_cell();
let rest_mops_cell = allocator.alloc_common_range_cell();
```



```
Definition isbit c := forall i, mtable_values c i = 0 \/ mtable_values c i = 1.
Definition iscommon c := forall i, 0 <= mtable_values c i < common.

Axiom enabled_bit : isbit enabled_cell.
Axiom rest_mops_common : iscommon rest_mops_cell.
```

Machine Model: Other Axioms

Integer Overflow

Proofs often treat the numbers as integers rather than integers modulo *p*.

- Saves proof effort when it is clear from inspection that the values in the gates are so small that they will not overflow.
- We model the modulo operation precisely for the "encoding" functions that packs multiple values into one integer, and for OpBinModel.v since arithmetic operations require being particularly careful about overflow.

Axioms about Frontend and Prover

3 axioms about the size of the integers used in the prover, about the address of call instructions, and about two static call stack entries created by Wasmi.

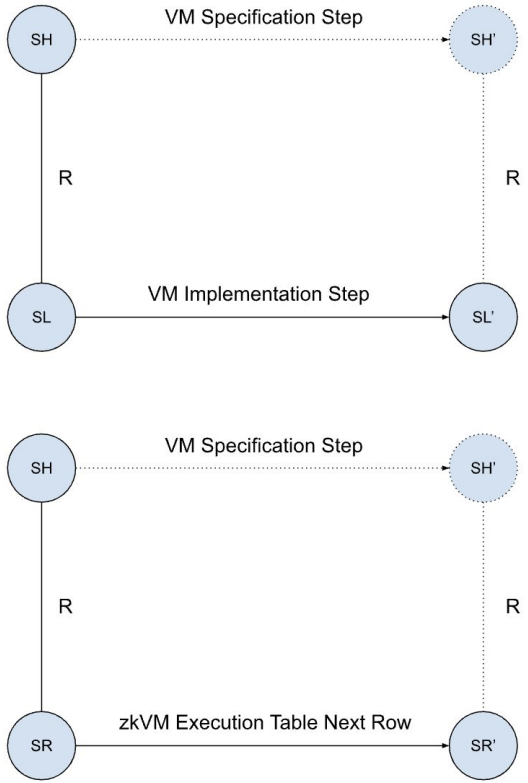
Top-level specification of instructions

The specification of a zkVM is similar to how you specify the verification of a conventional bytecode VM.

- Need a definition of the **abstract state** of the VM.
- A definition of the **steps** of the abstract machine.
- A **relation** to the concrete VM state (for a VM) or the table contents (for a zkVM)

Putting the step lemmas together you get the final theorem:

The *n*th row of the ETable represents the *n*th abstract state of a Wasm program execution.



Top-level specification of instruct

The specification of a zkVM is similar to how you sp
verification of a conventional bytecode VM.

- Need a definition of the **abstrac**
- A definition of the **steps** of
- A **relation** to the concrete VM state (for a VM) c
  table contents (for a zkVM)

Putting the step lemmas together you get the final the
The *n*th row of the ETable represents the *n*th ab
state of a Wasm program execution.

The definition of abstract state and steps are correctness critical and trusted!

We don't want to define these from scratch because we might make mistakes, instead we reuse existing definitions from the third-party **WasmCert** development. (Which has been validated in various ways.)

We can't literally reuse the WasmCert step definition because we are proving Wasmi bytecode rather than Wasm source, but we use all their data definitions and helper functions.

```
Record WasmState := {
    wasm_pc : label; (* program counter *)
    wasm_stack : list Z;
    wasm_globals : list Wasm.datatypes.global;
    wasm_memory : Wasm.datatypes.memory;
    wasm_callstack : list label
}.
```

Top-level specification of instr

The specification of a zkVM is similar to how you s
verification of a conventional bytecode VM.

- Need a definition of the **abstract state** of the
- A definition of the **steps** of the abstract mach
- A **relation** to the concrete VM state (for a VM
  table conten

Putting the step lemmas togeth

The *n*th row of the ETable repre
state of a Wasm program execution.

The data representation relation is 150 lines of Coq.

The program counter represented by two columns in the etable (for function id, and instruction id).

Stack, globals, heap, and callstack requires a specification of the MTable and JTable, we will come back to it.

```
(* "The i:th row of etable represents the state st". *)
Record state_rel (i : Z) (st : WasmState) := {
    state_pc_rel : (wasm_pc st) = (etable_values fid_cell i,
                                   etable_values iid_cell i);
    state_stack_rel : stack_rel (stk_map (etable_values eid_cell i))
                                (etable_values sp_cell i + 1)
                                (wasm_stack st);
    state_globals_rel : globals_rel (globals_map (etable_values eid_cell i))
                                    (wasm_globals st);
    state_heap_rel  : heap_rel  (heap_map (etable_values eid_cell i))
                                (etable_values mpages_cell i)
                                (etable_values maximal_memory_pages_cell i)
                                (wasm_memory st);
    state_callstack_rel :   callstack_rel (etable_values frame_id_cell i)
                                          (etable_values fid_cell i)
                                          (wasm_callstack st)


  }.
```

Top-level specification of instructions

E.g. the XOR instruction.
If the row *i* represents an abstract state *st*
and the instruction decoding bits says the current instruction is XOR
and the stack contains two values *x1 x2*
(the Wasm specification says "Assert: due to validation, two values of value type are on the top of the stack.")
Then the row *i+1* has pushed the value (*xor x1 x2*).
(Note that we are reusing the WasmCert definitions here.)

```
Theorem BitOp_Xor_correct : forall i st x1 x2 xs,
  0 <= i ->
  (etable_values enabled_cell i) = 1 ->
  mops_at_correct i ->
  etable_values (ops_cell BinBit) i = 1 ->
  etable_values op_class i = RTableModel.BitOp_Xor ->
  state_rel i st ->
  wasm_stack st = (Wasm_int.Z_of_uint i64m x1:: Wasm_int.Z_of_uint i64m x2::xs) ->
  state_rel (i+1) (update_stack (incr_iid st) (Wasm_int.Z_of_uint i64m (Wasm_int.int_xor i64m x2 x1) :: xs)).
```

The Proofs

The specifications are similar to a interpreted bytecode VM, but when doing the proofs, a zkVM has two main complications.

- **Arithmetic**. zk-provers natively work with integers modulo a large prime number, while the Wasm deals with 32- or 64-bit integers.
  This affects all the proofs, and is a major part of the work, but it is “local”.
- **Memory**. A conventional VM represents memory and control stack as mutable data structures. The **proof** matches two abstract/concrete updates.

But zkWasm handles this variable-sized data through the MTable and JTable…

```
func opMstore(pc *uint64, interpreter *EVMInterpreter, scope *ScopeContext) ([]byte, error) {
        // pop value of the stack
        mStart, val := scope.Stack.pop(), scope.Stack.pop()
        scope.Memory.Set32(mStart.Uint64(), &val)
        return nil, nil
}
```

The zkWasm Memory Table

Each memory write from the trace is recorded in the MTable, together with the start and end of its valid range.

```
int balance, amount;
void main () {
      balance = 100;
      amount = 10;
      balance -= amount; // withdraw
}
```

Execution Table

EID	OPCODE	STORE_ADDR	STORE_VALUE	LOAD_ADDR	LOAD_VALUE	...
1	STORE	balance	100			
2	STORE	amount	10			
3	LOAD			amount	10	
4	LOAD			balance	100	
5	SUB					
6	STORE	balance	90			

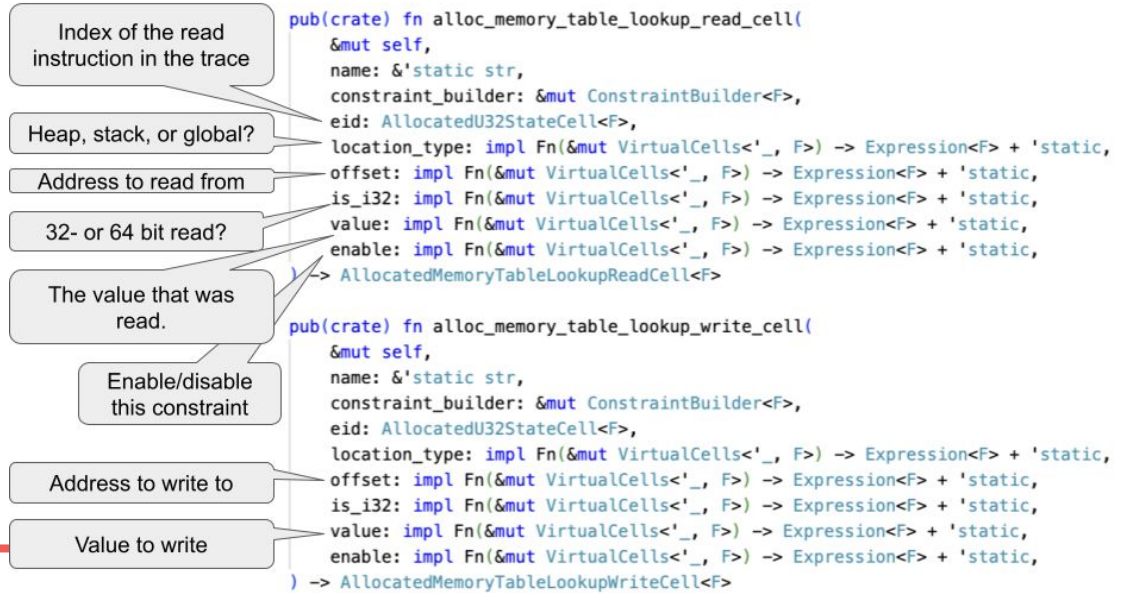
Memory Table

START_EID	END_EID	TYPE	ADDRESS	VALUE
1	6	global	balance	100
6	6	global	balance	90
2	6	global	amount	10

zkWasm Memory Abstraction Layer

zkWasm provides an *abstraction layer* in the form of two convenience functions `alloc_…read/write_cell`. The programmer can think of the memory table as a mutable structure. E.g. there are calls to this in the Memory Store circuits.

```
let memory_table_lookup_heap_write1 = allocator
    .alloc_memory_table_lookup_write_cell_with_value(
        "store write res1",
        constraint_builder,
        eid,
        move |____| constant_from!(LocationType::Heap as u64),
        move |meta| load_block_index.expr(meta),
        move |____| constant_from!(0),
        move |____| constant_from!(1),
    );
```



Specifying the Memory Abstraction Layer

In the proofs, we reify the intuition that the MTable “represents” a mutable dictionary, by writing a function memory_at that computes an *actual* dictionary from the table data, at a given *eid.*

We then prove that the abstraction layer functions are correct with respect to the dictionary.

- 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
  ```

- And 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
  ```

…Almost!

The zkWasm counting scheme

What if an attacker inserted a spurious row into the MTable? The witnesses are under the control of the attacker.

Execution Table of A Hack

EID	OPCODE	STORE_ADDR	STORE_VALUE	LOAD_ADDR	LOAD_VALUE	...
1	STORE	balance	100			
2	STORE	amount	10			
3	LOAD			amount	10	
4	LOAD			balance	10	
5	SUB					
6	STORE	balance	100			

Memory Table of A Hack

START_EID	END_EID	TYPE	ADDRESS	VALUE
1	3	global	balance	100
3	6	global	balance	110
6	6	global	balance	100
2	6	global	amount	10

The zkWasm Counting Scheme

The ETable has a column counting how many writes that instruction is *meant* to do.

The MTable has a count of the number of rows in the table.

There is an *equality constraint* between the the top cells (here, 3=3).

Intuition: if MTable had an extra row, the counts would not match.

Execution Table

EID	OPCODE	STORE_ADDR	STORE_VALUE	LOAD_ADDR	LOAD_VALUE	MOPS	...
1	STORE	balance	100			3	
2	STORE	amount	10			2	
3	LOAD			amount	10	1	
4	LOAD			balance	100	1	
5	SUB					1	
6	STORE	balance	90			1	

Memory Table

START_EID	END_EID	TYPE	ADDRESS	VALUE	MOPS
1	6	global	balance	100	3
6	6	global	balance	90	2
2	6	global	amount	10	1

The zkWasm Counting scheme

The correct specification of the alloc_read/write_cell functions is:

> If the constraints
> `(memory_table_lookup_write_cell eid type offset value)`
> hold and so do
> **`(mops_at eid type) = 1`**
> then `(memory_at(eid+1) type) = set (memory_at eid type) offset value`

where mops_at is the number of entries in the MTable at that *eid.*

Then we need to prove that, if the total count is correct for the first row in the ETable, then mops_at is correct for *each* row in the ETable.

The zkWasm Counting scheme

The correct specification of the alloc_read/write_cell functions is:

If the constraints
`(memory_table_lookup_write_cell eid type offset value)`
hold and so do
`(mops_at eid type) = 1`
then `(memory_at(eid+1) type) = set (m`
`offset value`

where mops_at is the number of entries in the MTabl

This is somewhat subtle! We need to prove that each instruction creates *at least* as many entries as expected, to show that it creates *at most* as many entries as expected. (See our blog post for details.)

Then we need to prove that, if the total count is correct for the first row in the ETable, then mops_at is correct for *each* row in the ETable.

A Tale of Two Bugs 1: Load8 Data Injection

Wasm has instructions Load64, Load32, Load16 and Load8 to read from heap.zkWasm includes constraints forcing the high bits to be zero.

```
constraint_builder.push(
    "op_load pick value size check",
    Box::new(move |meta| {
        vec![
            is_four_bytes.expr(meta)
                * (load_picked.u16_cells_le[2].expr(meta)
                    + load_picked.u16_cells_le[3].expr(meta)),
            is_two_bytes.expr(meta)
                * (load_picked.expr(meta) - load_picked_leading_u16.expr(meta)),
            is_one_byte.expr(meta) * (load_picked_leading_u16_u8_high.expr(meta)),
        ]
    }),
);
```

A Tale of Two Bugs 1: Load8 Data Injection

Wasm has instructions Load64, Load32, Load16 and Load8 to read from heap.zkWasm includes constraints forcing the high bits to be zero.

Should say:

```
is_one_byte.expr(meta)
    * (load_picked.expr(meta) - load_picked_leading_u16_u8_low.expr(meta)),
```

```
ze check",

.expr(meta)
    * (load_picked.u16_cells_le[2].expr(meta)
        + load_picked.u16_cells_le[3].expr(meta)),
is_two_bytes.expr(meta)
    * (load_picked.expr(meta) - load_picked_leading_u16.expr(meta)),
is_one_byte.expr(meta) * (load_picked_leading_u16_u8_high.expr(meta)),
        ]
    }),
);
```

Simple typo, in one of 600 lines of LOAD constrains, in 6000 lines of circuits. Arbitrary data injection.

Found through audit. Could also be found by FV or by fuzzing.

A Tale of Two Bugs (2): Faked Returns

The JTable design is almost the same as MTable, except both calls and returns are counted.

Same intuition: if there is an extra row in the the table, the counts will not match.

…right?

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

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

E-Table

execution step	call frame	function	instruction id	code	jops
0	0	buy_token	0	add_token()	1
1	1	add_token	0	token = token + 1	0
2	1	add_token	1	return	1
3	0	buy_token	1	…	0

J-Table

call frame	caller frame	caller function	caller's next instruction id	jops
1	0	buy_token	1	2

A Tale of Two Bugs (2): Faked Returns

It turns out that this scheme is not sound: you can cleverly create several extra *return* instructions in the ETable.

Solution: count calls separately from returns.

Missed in auditing. Seems very hard to find with fuzzing, but is found with FV.

E-Table

execution step	call frame	function	instruction id	code	jops
0	0	buy_token	0	add_token()	1
1	1	add_token	0	token = token + 1	0
2	1	add_token	1	return	1
3	1	add_token	0	token = token + 1	0
4	1	add_token	1	return	1
5	1	add_token	0	token = token + 1	0
6	1	add_token	1	return	1
7	0	buy_token	1	…	0

J-Table

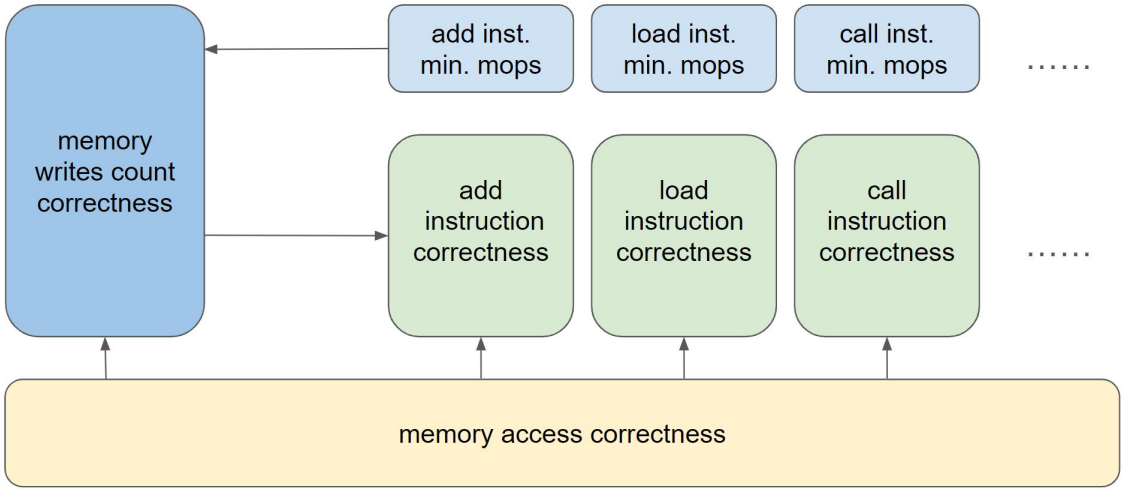
call frame	caller frame	caller function	caller's next instruction id	jops
1	0	buy_token	1	2
1	1	add_token	0	2

zkWasm Project Experience

About 6 month in total. We followed CertiK's usual approach of "audit-led formal verification", where the entire project (not just circuits) was audited, which builds familiarity for the FV work.

We started with a small team to develop the specifications, and then added more verification engineers to do the per-instruction proofs.

This requires some care to make the specifications modular, particularly the MTable.



Outlook for verifying a zkRISCV

Can we leverage this experience to verify a RISC-V zkVM? In phase one, our proposed work is exploratory, trying set out a solid foundation:

- Cost-efficient verification of the zkVM circuits.
  - Verifying all of the circuits is not mandatory.
- Flexibility in ISA extensions, arithmetization, and circuit implementation styles.
  - Lessons should be applicable to multiple zkVMs.
- Clear and precise trusted verification base, with paths to improving it further.
- Collaborative/sustainable engineering.
  - Third parties should be able to do part of the work, and to keep maintaining it indefinitely.

Therefore: pick *one* zkRISCV implementation (e.g. Jolt), *completely* specify and and *partially prove* it, then use this as a basis of comparison to other implementations.

Reusing the zkWasm experience for de-risking

Wasm is slightly more complicated than RISC-V, so this gives confidence that it can be done.

- Coq Machine Model (postulated tables and Axiom like zkWasm)
  - This gives us confidence in the cost estimates
  - Still will require research to handle the differences between Halo2 and Jolt eDSL.
- Specification should be at similar level of abstraction as Wasm
  - This means not re-using SAIL model directly, though in future work we can prove equivalence
  - Top-level theorems can include completeness as well as soundness, to get a cost estimate.
- Detailed code review/specifications of the theorems, before scaling up to proving (like zkWasm)
  - This order minimizes risk. And in future, per-instruction proofs can be posted as bounties.

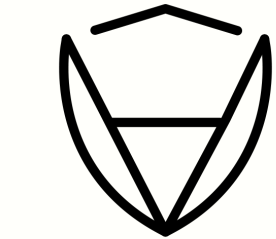
The resulting zkRISCV artifact is a model for Phase 2

We will have the complete set of specifications, and enough experience to be confident in the verification cost of the instructions that remain.

We use this experience to plan out similar verifications for other zkRISCV implementations.

A clear set of directions for phase2 efforts:

- Completing the proofs, via more contracts and/or bounties.
- Reducing trusted base by connecting to other verified components (verified REVM, machine-level RISC-V spec, verified zk-prover).



Questions?