

Verifying Proofs on Blockchain

Jeremy Avigad, Carnegie Mellon

Lior Goldberg, StarkWare Industries

David Levit, StarkWare Industries

Yoav Seginer, independent

Alon Titelman, StarkWare Industries

February 3, 2025

Perspectives on proof

- Proof theory
 - reductive proof theory
 - structural proof theory
 - proof complexity
- Proof mining
 - computational information
 - explicit bounds (e.g. on rates of convergence)
- History of mathematics
 - language, assumptions, rules of inference
- Philosophy of mathematics
 - How is understanding conveyed in mathematical proof?
 - What norms and values are evident?

Perspectives on proof

- Interactive theorem proving
 - proof languages
 - type class inference
 - tactics and metaprogramming languages
- Automated theorem proving
 - decision procedures
 - search procedures
 - verification and proof reconstruction
- Cryptographic proof
 - digital signatures
 - interactive proof systems
 - zero knowledge proofs

Perspectives on proof

What do proofs do?

- convince
- save time
- communicate additional information
- communicate ideas, methods, understanding.

What this talk is about

StarkWare has developed means of “running” decentralized applications (dapps) on blockchain more efficiently.

- They use this to run a cryptocurrency exchange.
- They have a platform for developers to run their applications.

Both use a family of programming languages, [Cairo](#), that they have developed.

We have been using Lean to verify their methods and code.

What this talk is about

More precisely, we are using the Lean proof assistant to verify the soundness (and completeness) of cryptographic proof procedures.

Two notions of proof:

- formal proofs in Lean
- cryptographic proofs based on Cairo

Outline

- Blockchain, smart contracts, and Cairo
- Interactive proof assistants and Lean
- Cairo verification
- Verifying the encoding of Cairo execution traces
- A proof-producing compiler for CairoZero
- Verifying elliptic curve computations

Blockchain

The magic:

- A ledger in the sky.
- The ledger grows, but nothing is ever deleted.
- Users can own resources, like bitcoin, and transfer them.

Application: banking without government or institutional control (or oversight).

Under the hood:

- Decentralized maintenance.
- Proof of work or proof of stake.
- Clever incentives to maintain and extend the blockchain.
- Cryptography.

Smart contracts

The magic:

- One can put a computer program on the ledger that executes when some conditions are met.

Applications: contracts, auctions, exchanges, trading NFTs, selling concert tickets, voting, etc., again without government or institutional control.

Under the hood:

- Suitable programming languages.
- Blockchain.

Smart contracts

Limitations:

- It is expensive. (Everyone maintaining the blockchain has to execute the code.)
- It is slow. (For example, the number of transactions that Visa performs in a second dwarf what can be done on blockchain.)

Developers seek “layer-2 solutions,” which move computation off chain and minimize what has to be computed on chain.

StarkWare uses ideas from complexity theory (interactive proof systems) to great effect.

Proof compression

StarkWare's approach:

- Design a suitable programming language.
- Write programs in such a way that successful completion guarantees the result (of the exchange, auction, etc.) is correct.
- Use a cryptographic protocol to publish a short (probabilistic) proofs that programs run to completion.

Outline

- Blockchain, smart contracts, and Cairo
- Interactive proof assistants and Lean
- Cairo verification
- Verifying the encoding of Cairo execution traces
- A proof-producing compiler for CairoZero
- Verifying elliptic curve computations

Formal methods

Formal methods are a body of logic-based methods used in computer science to

- write specifications (for hardware, software, protocols, and so on), and
- verify that artifacts meet their specifications.

They rely on:

- formal languages
- formal semantics
- formal rules of inference.

Interactive theorem provers

We have known since the early twentieth century that mathematics can be formalized.

With the help of computational proof assistants, this can be carried out in practice.

In many systems, the formal proof can be extracted and verified independently.

Interactive theorem provers

Proof assistants are now used for

- hardware, software, and systems verification
- mathematics and the mathematical sciences.

Some proof assistants for mathematics:

- Mizar (1973, set theory)
- Isabelle (1986, simple type theory)
- Rocq (1989, dependent type theory)
- HOL Light (1994, simple type theory)
- Lean (2013, dependent type theory)

Lean and Mathlib

Very few mathematicians were using formal methods in 2017.

Things have changed dramatically since then:

- [Mathlib](#) has almost 1.6 million lines of code.
- The [Lean Zulip channel](#) has 11K members, about 850 active in any two-week period.
- There have been a number of celebrated successes.
- There have been a number of articles in the general media.
- There are [meetings, workshops, and summer schools](#) related to Lean.
- There is growing interest and enthusiasm in the mathematical community.

Outline

- Blockchain, smart contracts, and Cairo
- Interactive proof assistants and Lean
- Cairo verification
- Verifying the encoding of Cairo execution traces
- A proof-producing compiler for CairoZero
- Verifying elliptic curve computations

Cairo

I am a powerful prover, and you are a resource-limited verifier.

- We agree on a program in a programming language, Cairo, such that successful termination guarantees my claim.
- A compiler translates it to assembly code and then machine code for the Cairo CPU.
- The claim that there is an assignment to memory, extending the program and input data, such that the CPU runs to completion is encoded as the existence of solutions to a parametric family of polynomials.
- The Cairo runner publishes to blockchain short cryptographic certificates that guarantee the existence of the solutions.

Cairo

Should you trust it?

- You have to believe the Cairo program does what it is supposed to.
- You have to believe that the translation to machine code is correct.
- You have to believe that the polynomial encoding is correct.
- You have to believe the cryptographic protocol.

We have addressed the first three.

Others have verified cryptographic protocols.

Cairo verification

Our approach:

- We have proved, in Lean, the correctness of the algebraic encoding of execution traces.
- We have built a verifying compiler for CairoZero that enables us to prove, in Lean, that the machine code satisfies high-level specifications.
- We have verified the correctness of important high-level code, for example, code for validating signatures with elliptic curves.
- We trust the cryptographic protocol at the bottom.

Cairo verification

More precisely, *Cairo* is an architecture and a family of languages:

- the Cairo virtual CPU and instruction set
- Cairo assembly language, *Casm*
- a programming language, *CairoZero*, that adds variables, structures, function definitions, etc.
- a higher-level programming language, *Cairo*, that includes type safety guarantees.

Cairo verification

The successful completion of a Cairo program is meant to *prove* something.

- *Soundness*: successful completion implies that the prover knows an assignment to memory that guarantees that a property holds.
- *Completeness*: if the property holds, the prover can assign values to memory to ensure successful completion.

This talk is about proving soundness of programs written in CairoZero.

Cairo verification

References:

- A public repository, <https://github.com/starkware-libs/formal-proofs>
- A verified algebraic representation of Cairo program execution, CPP 2022.
- A proof-producing compiler for blockchain applications, ITP 2023.
- A proof-producing compiler for blockchain applications (arXiv, to appear in the Journal of Automated Reasoning).
- The latest programming language, Cairo.

Cairo verification

Other projects:

- Verifying soundness and completeness of Cairo library functions (“libfuncs”).
- Verifying other proof schemes.

So there is more to come!

Outline

- Blockchain, smart contracts, and Cairo
- Interactive proof assistants and Lean
- Cairo verification
- Verifying the encoding of Cairo execution traces
- A proof-producing compiler for CairoZero
- Verifying elliptic curve computations

Cryptographic protocols

SNARK = Succinct Non-interactive ARgument of Knowledge

STARK = Scalable Transparent ARgument of Knowledge

zk-SNARK = zero-knowledge SNARK, zk-STARK =
zero-knowledge STARK

From the Avenger movies



The algebraic intermediate representation (AIR)

Fix a large prime number p , and let F_p be the finite field of integers modulo p .

A STARK lets me convince you that I have a table of field elements, satisfying some constraints.

For polynomials $P_1(\vec{x}_1), \dots, P_s(\vec{x}_s)$ and periodic sets D_1, \dots, D_s , the constraints say that for every $j = 1, \dots, s$ and every $r \in D_j$, we have $P_k(\vec{x}_k^{(r)}) = 0$, where $\vec{x}_k^{(r)}$ is \vec{x}_k shifted by r rows.

From the AIR to executions

I want to convince you that there exists a full assignment to memory extending the partial one and an execution trace from the start state to the final state.

STARK magic will convince you that I possess a table of data, some of which you can see, that satisfies a bunch of polynomials over a large finite field.

Steps:

From the AIR to executions

I want to convince you that there exists a full assignment to memory extending the partial one and an execution trace from the start state to the final state.

STARK magic will convince you that I possess a table of data, some of which you can see, that satisfies a bunch of polynomials over a large finite field.

Steps:

- Encode statements about execution traces in terms of polynomials. (The AIR.)

From the AIR to executions

I want to convince you that there exists a full assignment to memory extending the partial one and an execution trace from the start state to the final state.

STARK magic will convince you that I possess a table of data, some of which you can see, that satisfies a bunch of polynomials over a large finite field.

Steps:

- Encode statements about execution traces in terms of polynomials. (The AIR.)
- Prove to you that the encoding is correct. (The whitepaper.)

From the AIR to executions

I want to convince you that there exists a full assignment to memory extending the partial one and an execution trace from the start state to the final state.

STARK magic will convince you that I possess a table of data, some of which you can see, that satisfies a bunch of polynomials over a large finite field.

Steps:

- Encode statements about execution traces in terms of polynomials. (The AIR.)
- Prove to you that the encoding is correct. (The whitepaper.)
- Verify the proof. (Lean.)

CPU semantics

The CPU state consists of three registers: a program counter, an allocation pointer, and a frame pointer.

Instructions:

- Assert two things are equal (e.g. $[x] = [y] + 3$).
- Jump.
- Conditional jump.
- Call.
- Return.

Memory consists of values in a large finite field. Arithmetic operations are $+$ and $*$.

CPU semantics

We have:

- a whitepaper specification of the CPU
- a formal specification in Lean

Formally, we define the next state relation:

```
def next_state (mem : F → F) (s t : register_state F) :  
  Prop := ...
```

There may be no next state (when assertions fail) or multiple next states (undefined behavior).

CPU semantics

An instruction consists of 15 one-bit flags and three 16-bit bitvectors.

Flags determine the relevant instruction, memory addressing modes, and so on.

```
structure register_state (F : Type*) :=
  (pc : F) (ap : F) (fp : F)

def next_fp : option F :=
  match i.opcode_call, i.opcode_ret, i.opcode_assert_eq with
  | ff, ff, ff := some s.fp
  | tt, ff, ff := some (s.ap + 2)
  | ff, tt, ff := some (i.dst mem s)
  | ff, ff, tt := some s.fp
  | _, _, _ := none
end
```

The polynomial constraints

The polynomial constraints need to guarantee the existence of a valid trace consistent with the public data.

- Unpacking and interpreting instructions.
- Verifying that certain field elements are integers, and in range.
- Verifying that the memory assignment is valid, and extends the public one.
- Verifying each step.

These are found in three places:

- Locally, in each file.
- Globally, collected together.
- In a slightly different format, autogenerated from StarkWare code.

The polynomial constraints

The raw polynomial constraints are generated from the same code used by the StarkWare compiler to generate STARK certificates.

Clients can verify from the published certificates that these are the polynomials that were used.

The Lean proof verifies that the existence a table of values satisfying these polynomials implies the existence of the relevant execution trace.

The polynomial constraints

```
structure cpu__update_registers (inp : input_data F) (c : columns F) :
  Prop :=
(update_ap__ap_update :  $\forall i: \text{nat}, ((i \% 16 = 0)) \wedge \neg (i = 16 * (\text{inp.trace\_length} / 16 - 1))) \rightarrow (c.\text{column21}.\text{off } i 16) - ((c.\text{column21}.\text{off } i 0) + ((c.\text{column1}.\text{off } i 10) - ((c.\text{column1}.\text{off } i 11) + (c.\text{column1}.\text{off } i 11)))) * (c.\text{column21}.\text{off } i 12) + (c.\text{column1}.\text{off } i 11) - ((c.\text{column1}.\text{off } i 12) + (c.\text{column1}.\text{off } i 12)) + ((c.\text{column1}.\text{off } i 12) - ((c.\text{column1}.\text{off } i 13) + (c.\text{column1}.\text{off } i 13))) * 2) = 0)
(update_fp__fp_update :  $\forall i: \text{nat}, ((i \% 16 = 0)) \wedge \neg (i = 16 * (\text{inp.trace\_length} / 16 - 1))) \rightarrow (c.\text{column21}.\text{off } i 24) - ((1 - ((c.\text{column1}.\text{off } i 12) - ((c.\text{column1}.\text{off } i 13) + (c.\text{column1}.\text{off } i 13)) + (c.\text{column1}.\text{off } i 13) - ((c.\text{column1}.\text{off } i 14) + (c.\text{column1}.\text{off } i 14)))) * (c.\text{column21}.\text{off } i 8) + ((c.\text{column1}.\text{off } i 13) - ((c.\text{column1}.\text{off } i 14) + (c.\text{column1}.\text{off } i 14))) * (c.\text{column19}.\text{off } i 9) + ((c.\text{column1}.\text{off } i 12) - ((c.\text{column1}.\text{off } i 13) + (c.\text{column1}.\text{off } i 13))) * ((c.\text{column21}.\text{off } i 0) + 2)) = 0)
(update_pc__pc_cond_negative :  $\forall i: \text{nat}, ((i \% 16 = 0)) \wedge \neg (i = 16 * (\text{inp.trace\_length} / 16 - 1))) \rightarrow (1 - ((c.\text{column1}.\text{off } i 9) - ((c.\text{column1}.\text{off } i 10) + (c.\text{column1}.\text{off } i 10)))) * (c.\text{column19}.\text{off } i 16) + (c.\text{column21}.\text{off } i 2) * ((c.\text{column19}.\text{off } i 16) - ((c.\text{column19}.\text{off } i 0) + (c.\text{column19}.\text{off } i 13))) - ((1 -$$$ 
```

The main theorem: first cut

Given

- a partial assignment to memory,
- a start state,
- a final state,
- a table of data satisfying the polynomial constraints with respect to these,

there exists

- a total assignment to memory extending the partial one and
- a valid execution trace from the start state to the final state.

A challenge

Suppose at a stage in the trace, the code asserts that memory position a_i contains value v_i .

To establish the the execution succeeds, the prover needs to establish two things:

- All the pairs (a_i, v_i) form a consistent memory assignment.
- The assignment extends the public partial assignment that the verifier agreed to.

This can't be done efficiently with polynomial constraints, which are *local* constraints.

A similar problem: verifying that a field element is a cast of an integer in a certain range.

A solution

The protocol uses yet another cryptographic solution.

Within the STARK, the prover commits to 23 columns of data.

A cryptographic hash is generated, roughly, a random challenge.

The prover responds with two more columns of data.

We prove that with high probability, the fact that the data satisfies the constraints implies the desired conclusion.

The main theorem

Given

- a partial assignment to memory,
- a start state,
- a final state,
- a table of data,

there exist small sets B_1, B_2, B_3 , such that given

- field elements x_1, x_2, x_3 ,
- two more columns of data,

if the x s are not in the B s and the data satisfies the polynomial constraints, there exists

- a total assignment to memory extending the partial one and
- a valid execution trace from the start state to the final state.

The main theorem

```
theorem final_correctness
  {F : Type} [field F] [fintype F]
  (char_ge : ring_char F ≥ 2^63)
  /- public data -/
  (inp : input_data F)
  (pd  : public_data F)
  (pc  : public_constraints inp pd)
  (c   : columns F) :
  /- sets to avoid -/
  ∃ bad1 bad2 bad3 : finset F,
  bad1.card ≤ (inp.trace_length / 2)^2 ∧
  bad2.card ≤ inp.trace_length / 2 ∧
  bad3.card ≤ inp.trace_length ∧
```

The main theorem

```
∀ ci : columns_inter F,  
  /- autogenerated constraints -/  
  cpu__decode c ∧  
  cpu__operands c ∧  
  cpu__update_registers c inp ∧  
  cpu__opcodes c ∧  
  memory inp pd c ci ∧  
  rc16 inp pd c ci ∧  
  public_memory c ∧  
  initial_and_final inp c ∧  
  /- probabilistic constraints -/  
  pd.hash_interaction_elm0 ∉ bad1 ∧  
  pd.interaction_elm ∉ bad2 ∧  
  pd.interaction_elm ≠ 0 ∧  
  pd.rc16__perm__interaction_elm ∉ bad3 →
```

The main theorem

```
/- the conclusion -/  
let T := inp.trace_length / 16 - 1 in  
∃ mem : F → F,  
  option.fn_extends mem inp.m_star ∧  
  ∃ exec : fin (T + 1) → register_state F,  
    (exec 0).pc = inp.initial_pc ∧  
    (exec 0).ap = inp.initial_ap ∧  
    (exec 0).fp = inp.initial_ap ∧  
    (exec (fin.last T)).pc = inp.final_pc ∧  
    (exec (fin.last T)).ap = inp.final_ap ∧  
  ∀ i : fin T,  
    next_state mem (exec i.cast_succ)  
      (exec i.succ)
```

Outline

- Blockchain, smart contracts, and Cairo
- Interactive proof assistants and Lean
- Cairo verification
- Verifying the encoding of Cairo execution traces
- A proof-producing compiler for CairoZero
- Verifying elliptic curve computations

Cairo verification

Programs are used to *prove* claims to a skeptical verifier. To prove $f(x) = y$, write a program that computes $f(x)$, compares it to y , and fails if they don't agree.

Consider the instruction $y = x + 5$.

You can think of this as a hint to the prover saying: “make sure you put the value of $x + 5$ in the memory location assigned to y .”

From a soundness perspective, it says “the prover has assigned values to memory making $y = x + 5$.”

Cairo verification

Novelties:

1. The fact that the programming language is used to prove things introduces some quirks (e.g. memory is read only).
2. The cryptographic foundation introduces some quirks (e.g. all values are elements of a large finite field).
3. From the high-level specification down to the table of polynomials, our proofs are verified entirely in Lean.
4. We harvest just enough information from the compiler to construct, automatically, long source-code proofs in Lean.
5. We provide flexible means to prove our own specifications from autogenerated ones.
6. The compiler developers barely even noticed that we were there.

The Cairo CPU

The CPU state consists of three registers: a program counter, an allocation pointer, and a frame pointer.

An instruction consists of 15 one-bit flags and three 16-bit bitvectors.

Instructions:

- Assert two things are equal (e.g. $[ap + 3] = [fp - 5] + 3$).
- Jump.
- Conditional jump.
- Call.
- Return.

Memory consists of values in a large finite field. Arithmetic operations are $+$ and $*$.

Semantics

We have a formal specification of the execution semantics.

```
def ensures (mem : F → F) (σ : register_state F)
  (P : ℕ → register_state F → Prop) : Prop :=
  ∀ n : ℕ, ∀ exec : fin (n+1) → register_state F,
  is_halting_trace mem exec → exec 0 = σ →
  ∃ i : fin (n + 1), ∃ κ ≤ i, P κ (exec i)
```

“Given memory mem and starting state σ , if execution runs to completion, then at some point P holds.”

Modulo the algebraic encoding, the cryptographic certificate verifies that the execution runs to completion.

For technical reasons, we sometimes need to know that some value is less than the length of the execution.

Modeling the assembly language in Lean

The compiler emits assembly code as well as the machine translations:

```
...
[ap] = [fp + (-4)]; ap++
[ap] = [fp + (-3)]; ap++
call rel -11
ret
```

We can also get it to emit hackish Lean notation that approximates the assembly syntax:

```
def starkware.cairo.common.math.code_assert_nn_le : list F := [
  ...
  'assert_eq['dst[ap] === 'res['op1[fp+ -4]];ap++].to_nat,
  'assert_eq['dst[ap] === 'res['op1[fp+ -3]];ap++].to_nat,
  'call_rel['op1[imm]].to_nat, -11,
  'ret[].to_nat ]
```

Modeling the assembly language in Lean

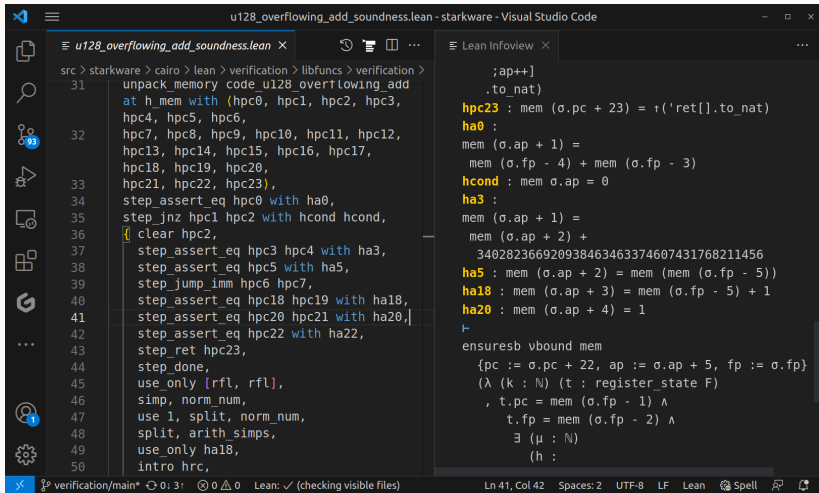
The Lean notation represents Lean definitions of machine code instructions.

We can evaluate them and compare them to the values output by the compiler.

So we are really proving things about the Cairo machine code.

We then wrote tactics that enable us to step through the machine code.

Modeling the assembly language in Lean



The image shows a screenshot of the Visual Studio Code editor. The main window displays a Lean file named `u128_overflowing_add_soundness.lean`. The code in the editor is as follows:

```
src > starkware > cairo > lean > verification > libfuncs > verification >
31  unpack_memory_code u128_overflowing_add
    at h_mem with (hpc0, hpc1, hpc2, hpc3,
32  hpc4, hpc5, hpc6,
    hpc7, hpc8, hpc9, hpc10, hpc11, hpc12,
    hpc13, hpc14, hpc15, hpc16, hpc17,
    hpc18, hpc19, hpc20,
33  hpc21, hpc22, hpc23),
34  step_assert_eq hpc0 with ha0,
35  step_jnz hpc1 hpc2 with hcond hcond,
36  { clear hpc2,
37    step_assert_eq hpc3 hpc4 with ha3,
38    step_assert_eq hpc5 with ha5,
39    step_jump_imm hpc6 hpc7,
40    step_assert_eq hpc18 hpc19 with ha18,
41    step_assert_eq hpc20 hpc21 with ha20,|
42  step_assert_eq hpc22 with ha22,
43  step_ret hpc23,
44  step_done,
45  use only [rfl, rfl],
46  simp, norm_num,
47  use 1, split, norm_num,
48  split, arith_simps,
49  use only ha18,
50  intro hrc,
```

The right-hand pane shows the Lean Infocview, displaying the current proof state:

```
;ap++]
.to_nat)
hpc23 : mem (σ.pc + 23) = ↑('ret[] .to_nat)
ha0 :
mem (σ.ap + 1) =
  mem (σ.fp - 4) + mem (σ.fp - 3)
hcond : mem σ.ap = 0
ha3 :
mem (σ.ap + 1) =
  mem (σ.ap + 2) +
  340282366920938463374607431768211456
ha5 : mem (σ.ap + 2) = mem (mem (σ.fp - 5))
ha18 : mem (σ.ap + 3) = mem (σ.fp - 5) + 1
ha20 : mem (σ.ap + 4) = 1
⊢
ensuresb vbound mem
{pc := σ.pc + 22, ap := σ.ap + 5, fp := σ.fp}
(λ (k : ℕ) (t : register_state F)
  , t.pc = mem (σ.fp - 1) ∧
  t.fp = mem (σ.fp - 2) ∧
  ∃ (μ : ℕ)
  (h :
```

The status bar at the bottom indicates the current position is Ln 41, Col 42, with 2 spaces, UTF-8 encoding, and LF line endings. The Lean compiler is currently checking visible files.

CairoZero

CairoZero code looks like this:

```
func assert_nn{range_check_ptr}(a) {  
    a = [range_check_ptr];  
    let range_check_ptr = range_check_ptr + 1;  
    return (); }
```

```
func assert_le{range_check_ptr}(a, b) {  
    assert_nn(b - a);  
    return (); }
```

```
func assert_nn_le{range_check_ptr}(a, b) {  
    assert_nn(a);  
    assert_le(a, b);  
    return (); }
```

There are also conditionals, structures, recursive calls, ...

Verifying CairoZero

We want to write specifications about the high-level CairoZero functions, and verify that the machine code meets the specifications.

Steps:

- Write a Python tool that extracts a (naive) high-level specification of each Cairo function, and *proves* that the compiled code satisfies the naive specification.
- For each particular program, show (iteratively) that the naive specifications imply our own specifications.

We have used this method to verify parts of the CairoZero library, including a validation procedure for cryptographic signatures.

Verifying CairoZero

Notes:

- Memory is read only; the CPU makes assertions about the contents of memory.
- Programs are used to *prove* claims to a skeptical verifier. To prove $f(x) = y$, write a program that computes $f(x)$, compares it to y , and fails if they don't agree.
- Memory locations contain elements of a field, but there are cryptographic primitives that allow us to say that x is the cast of an integer in $[0, 2^{128})$.
- In this phase, we only cared about *soundness*. Termination and memory management are free.

Outline

- Blockchain, smart contracts, and Cairo
- Interactive proof assistants and Lean
- Cairo verification
- Verifying the encoding of Cairo execution traces
- A proof-producing compiler for CairoZero
- Verifying elliptic curve computations

Elliptic curve computations

Any elliptic curve over a field of characteristic not equal to 2 or 3 can be described as the set of solutions to an equation $y^2 = x^3 + ax + b$, the so-called *affine* points, together with one additional *point at infinity*.

Elliptic curve computations

Assuming the curve is nonsingular, the set of such points has the structure of an abelian group, where the zero is defined to be the point at infinity and addition between affine points defined as follows:

- To add (x, y) to itself, let $s = (3x^2 + a)/2y$, let $x' = s^2 - 2x$, and let $y' = s(x - x') - y$. Then $(x, y) + (x, y) = (x', y')$. This is known as *point doubling*.
- $(x, y) + (x, -y) = 0$, that is, the point at infinity. In other words, $-(x, y) = (x, -y)$.
- Otherwise, to add (x_0, y_0) and (x_1, y_1) , let $s = (y_0 - y_1)/(x_0 - x_1)$, let $x' = s^2 - x_0 - x_1$, and let $y' = s(x_0 - x') - y_0$. Then $(x_0, y_0) + (x_1, y_1) = (x', y')$.

Elliptic curve computations

We verified functions that compute the scalar product $n \cdot x$, for:

- secp256k1: the curve $y^2 = x^3 + 7$ over the finite field of integers modulo the prime $p = 2^{256} - 2^{32} - 977$.
- secp256r1: $y^2 = x^3 - 3x + b$ over the finite field of integers modulo the prime $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$, where b is the big number below:

0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b

Cairo's underlying field has characteristic $p = 2^{251} + 17 \cdot 2^{192} + 1$.

Elliptic curve computations

Our Lean verification has to mediate between at least three different representations:

- Elements x of the secp field of integers modulo the secp prime number.
- Triples (i_0, i_1, i_2) of integers, suitably bounded, that represent such elements.
- Triples of elements (d_0, d_1, d_2) of the underlying field \mathbb{F} of the Cairo machine model, assumed or checked to be casts of such integers.

The code used optimization tricks.

Verification required a subtle bounds and careful side conditions.

Verifying CairoZero

```
// Given a scalar, an integer m in the range [0, 250), and a point on
// the elliptic curve, point, verifies that 0 <= scalar < 2**m and
// returns (2**m * point, scalar * point).
func ec_mul_inner{range_check_ptr}(point: EcPoint, scalar: felt,
                                   m: felt) -> (pow2: EcPoint, res: EcPoint)
{
    if (m == 0) {
        scalar = 0;
        let ZERO_POINT = EcPoint(BigInt3(0, 0, 0), BigInt3(0, 0, 0));
        return (pow2=point, res=ZERO_POINT);
    }

    alloc_locals;
    let (double_point: EcPoint) = ec_double(point);
    %{ memory[ap] = (ids.scalar % PRIME) % 2 %}
    jmp odd if [ap] != 0, ap++;
    return ec_mul_inner(point=double_point, scalar=scalar / 2, m=m - 1);

    ...
}
```

Verifying CairoZero

-- Do not change this definition.

```
def auto_spec_ec_mul_inner (mem : F → F) (κ : ℕ)
  (range_check_ptr : F) (point : EcPoint mem)
  (scalar m ρ_range_check_ptr : F)
  (ρ_pow2 ρ_res : EcPoint mem) : Prop :=
((m = 0 ∧
  scalar = 0 ∧
  ∃ ZERO_POINT : EcPoint mem, ZERO_POINT = {
    x := { d0 := 0, d1 := 0, d2 := 0 },
    y := { d0 := 0, d1 := 0, d2 := 0 }
  } ∧
  16 ≤ κ ∧
  ρ_range_check_ptr = range_check_ptr ∧
  ρ_pow2 = point ∧
  ρ_res = ZERO_POINT) ∨
(m ≠ 0 ∧
  ∃ (κ₁ : ℕ) (range_check_ptr₁ : F) (double_point : EcPoint mem),
  spec_ec_double mem κ₁ range_check_ptr point range_check_ptr₁
  double_point ∧
  ∃ anon_cond : F,
  ((anon_cond = 0 ∧ ...))))
```

Verifying CairoZero

```
-- You may change anything in this definition except the name and
-- arguments.
def spec_ec_mul_inner (mem : F → F) (κ : ℕ) (range_check_ptr : F)
  (point : EcPoint mem) (scalar m ρ_range_check_ptr : F)
  (ρ_pow2 ρ_res : EcPoint mem) : Prop :=
  ∀ (secpF : Type) [hsecp : secp_field secpF],
  point.x ≠ ⟨0, 0, 0⟩ →
  ∀ hpt : BddECPointData secpF point,
  ∃ nm : ℕ,
  m = ↑nm ∧
  nm < ring_char F ∧
  (nm ≤ SECP_LOG2_BOUND →
    ∃ scalarn : ℕ,
    scalar = ↑scalarn ∧
    scalarn < 2^nm ∧
    ∃ hpow2 : BddECPointData secpF ρ_pow2,
    ρ_pow2.x ≠ ⟨0, 0, 0⟩ ∧
    hpow2.toECPoint = 2^nm · hpt.toECPoint ∧
    ∃ hres : BddECPointData secpF ρ_res,
    hres.toECPoint = scalarn · hpt.toECPoint)
```


Verifying CairoZero

-- Do not change the statement of this theorem. You may change the proof.

```
theorem sound_ec_mul_inner
  {mem : F → F} (κ : ℕ)
  (range_check_ptr : F) (point : EcPoint mem)
  (scalar m ρ_range_check_ptr : F) (ρ_pow2 ρ_res : EcPoint mem)
  (h_auto : auto_spec_ec_mul_inner mem κ range_check_ptr point
    scalar m ρ_range_check_ptr ρ_pow2 ρ_res) :
  spec_ec_mul_inner mem κ range_check_ptr point scalar m
  ρ_range_check_ptr ρ_pow2 ρ_res :=
begin
  intros secpF _ ptxnez hpt,
  rcases h_auto with ⟨neq, scalareq, _, rfl, _, _, ret1eq, ret2eq⟩ |
  ⟨nnz, _, _, double_pt, hdouble_pt, _, heven_or_odd⟩,
  { use 0, split,
    { rw [neq, nat.cast_zero] }, split,
    { rw PRIME.char_eq, apply PRIME_pos },
    intro _,
    use 0, split,
    ... }
  ...
end
```

Verifying CairoZero

```
theorem auto_sound_ec_mul_inner
-- arguments
(range_check_ptr : F) (point : EcPoint F) (scalar m : F)
-- code is in memory at  $\sigma.pc$ 
(h_mem : mem_at mem code_ec_mul_inner  $\sigma.pc$ )
-- all dependencies are in memory
(h_mem_0 : mem_at mem code_nondet_bigint3 ( $\sigma.pc - 407$ ))
...
(h_mem_9 : mem_at mem code_fast_ec_add ( $\sigma.pc - 143$ ))
-- input arguments on the stack
(hin_range_check_ptr : range_check_ptr = mem ( $\sigma.fp - 11$ ))
(hin_point : point = cast_EcPoint mem ( $\sigma.fp - 10$ ))
(hin_scalar : scalar = mem ( $\sigma.fp - 4$ ))
(hin_m : m = mem ( $\sigma.fp - 3$ ))
-- conclusion
: ensures_ret mem  $\sigma$  ( $\lambda \kappa \tau, \exists \mu \leq \kappa,$ 
  rc_ensures mem (rc_bound F)  $\mu$  (mem ( $\sigma.fp - 11$ )) (mem $  $\tau.ap - 13$ )
    (spec_ec_mul_inner mem range_check_ptr point scalar m
      (mem ( $\tau.ap - 13$ )) (cast_EcPoint mem ( $\tau.ap - 12$ ))
      (cast_EcPoint mem ( $\tau.ap - 6$ )))) := ...
```

Verifying CairoZero

Summary:

- Our tool generates Lean descriptions of the machine code, which the user never has to see.
- It also generates naive specifications.
- Users write their own specifications, and prove their that they are implied by the naive ones.
- Later autogenerated specifications refer to the user specifications.
- Our tool automatically uses the user's theorems in end-to-end correctness proofs that the user never has to see.

The group law for elliptic curves

```
def add : ECPPoint F → ECPPoint F → ECPPoint F
| ZeroPoint      b           := b
| a              ZeroPoint   := a
| (AffinePoint a) (AffinePoint b) :=
  if axbx: a.x = b.x then
    if ayby: a.y = -b.y then
      -- a = -b
      ZeroPoint
    else
      have a.y = b.y, from eq_of_on_ec a.h b.h axbx ayby,
      have a.y ≠ 0,
      by { contrapose! ayby, rw ←[this, ayby, neg_zero] },
      let p := ec_double (a.x, a.y) in
      AffinePoint ⟨p.1, p.2, on_ec_ec_double a.h this⟩
  else
    let p := ec_add (a.x, a.y) (b.x, b.y) in
    AffinePoint ⟨p.1, p.2, on_ec_ec_add a.h b.h axbx⟩
```

The group law for elliptic curves

Our formalization of the secp elliptic curves initially had one `sorry` each.

The group law for elliptic curves

```
instance : add_comm_group (ECPoint F) :=
{ add      := ECPoint.add,
  neg      := ECPoint.neg,
  zero     := ECPoint.ZeroPoint,
  add_assoc := sorry,
  zero_add := by { intro a, cases a; simp [ECPoint.add] },
  add_zero := by { intro a, cases a; simp [ECPoint.add] },
  add_left_neg := ECPoint.add_left_neg,
  add_comm  := ECPoint.add_comm }
```

The group law for elliptic curves

As we were finishing the secp256k1 verification, David Angdinata and Junyan Xu verified the group law in Lean, in great generality.

The group law for elliptic curves

```
def curve_to_ECPoint : (@curve F _).point → ECPoint F
| point.zero := ECPoint.ZeroPoint
| (@point.some _ _ _ x y h) :=
  ECPoint.AffinePoint ⟨x, y, on_ec_of_nonsingular h⟩

def ECPoint_to_curve : ECPoint F → (@curve F _).point
| ECPoint.ZeroPoint := point.zero
| (ECPoint.AffinePoint ⟨x, y, h⟩) :=
  point.some (nonsingular_of_on_ec h)

lemma left_inverse_curve_to_ECPoint :
  left_inverse (@curve_to_ECPoint F _) (@ECPoint_to_curve F _) :=
begin
  rintro (⟨⟩ | ⟨x, y, h⟩), { refl },
  simp [curve_to_ECPoint, ECPoint_to_curve]
end

theorem ECPoint_to_curve_add (a b : ECPoint F) :
  ECPoint_to_curve (a.add b) = ECPoint_to_curve a + ECPoint_to_curve b
:= ...
```


The group law for elliptic curves

```
instance : add_comm_group (ECPoint F) :=
{ add      := ECPoint.add,
  neg      := ECPoint.neg,
  zero     := ECPoint.ZeroPoint,
  add_assoc :=
    begin
      intros a b c,
      apply (left_inverse_curve_to_ECPoint).injective,
      simp [ECPoint_to_curve_add, add_assoc]
    end,
  zero_add := by { intro a, cases a; simp [ECPoint.add] },
  add_zero := by { intro a, cases a; simp [ECPoint.add] },
  add_left_neg := ECPoint.add_left_neg,
  add_comm  := ECPoint.add_comm }
```

Outline

- Blockchain, smart contracts, and Cairo
- Interactive proof assistants and Lean
- Cairo verification
- Verifying the encoding of Cairo execution traces
- A proof-producing compiler for CairoZero
- Verifying elliptic curve computations

Conclusions

- Cryptography is a kind of “applied proof theory,” largely about encoding claims and generating certificates.
- Interactive theorem is another kind of “applied proof theory,” about designing proof languages and proof systems that are practical for writing ordinary proofs.
- Blockchain verification is a good market for verification.
- It’s possible to do fun and interesting projects in an industrial setting.
- There are interesting logical and mathematical ideas involved.
- The applications provide new perspectives on proof.