A Proof-Producing Compiler for Blockchain Applications

Jeremy Avigad
Carnegie Mellon University, Pittsburgh, PA, USA

Lior Goldberg
StarkWare Industries Ltd., Netanya, Israel

David Levit
StarkWare Industries Ltd., Netanya, Israel

Yoav Seginer
Amsterdam, Netherlands

Alon Titelman
StarkWare Industries Ltd., Netanya, Israel

Abstract

Cairo is a programming language for running decentralized applications (dapps) at scale. Programs written in the Cairo language are compiled to machine code for the Cairo CPU architecture, and cryptographic protocols are used to verify the results of the execution traces efficiently on blockchain. We explain how we have extended the Cairo compiler with tooling that enables users to prove, in the Lean 3 proof assistant, that compiled code satisfies high-level functional specifications. We demonstrate the success of our approach by verifying primitives for computations with an elliptic curve over a large finite field, as well as their use in the validation of cryptographic signatures.

2012 ACM Subject Classification General and reference → Verification; Theory of computation → Logic and verification; Software and its engineering → Semantics

Keywords and phrases formal verification, smart contracts, interactive proof systems

Digital Object Identifier 10.4230/LIPIcs.ITP.2023.7

Supplementary Material *Software*: `https://github.com/starkware-libs/formal-proofs` archived at `swh:1:dir:6e3f3ad6721711a9e6dc1643d890edd6835e196e`

Acknowledgements We are grateful to the Lean developers and the Lean community for providing infrastructure for this project, and to three anonymous reviewers for numerous corrections and improvements.

1 Introduction

Cairo [16] is a programming language for running decentralized applications (dapps) at scale. Programs written in the Cairo language are compiled to machine code for the Cairo CPU architecture [14], which is run off chain by an untrusted prover. Using the STARK cryptographic proof system [6], the prover then publishes a succinct certificate for the result of the off-chain computation, which is verified efficiently on blockchain.

Here we describe an augmentation of the Cairo compiler that enables users to produce formal proofs that compiled machine code meets its high-level specifications. We retain enough information during the compilation phase for our verification tool to extract a description of the machine code as well as naive functional specifications of the source code. We automatically construct formal proofs, in the Lean 3 proof assistant [9], that the machine code meets these specifications. Users can then write their own specifications of the source code in Lean and prove that they are implied by the automatically generated ones. In doing so, they can make use of their specifications of functions earlier in the dependency chain. Using Lean to check both the user-written and autogenerated proofs yields end-to-end verification of the user's specifications, down to CPU semantics. In other work [4], we have moreover verified the correctness of the algebraic encoding of the CPU semantics that is used to generate the certificates used in the STARK protocol.

In Sections 2 to 4 we describe the Cairo CPU architecture, assembly code, and programming language, and we explain how we generate high-level specifications and construct the formal correctness proofs in Lean. Section 5 focuses on features of our work that are specific to the domain of application and the STARK encoding, namely, memory management in Cairo and mechanisms for computing with elements of a finite field. In Section 6, we demonstrate the success of our approach by describing our verification of elliptic curve computations over a large finite field and our verification of a procedure for validating digital signatures. In Section 7, we explain why our approach has been effective in practice, enabling us to verify production code without hindering the development of the compiler or the library. Our main contributions are therefore as follows:

- We provide means of obtaining end-to-end verification, in a foundational proof assistant, of Cairo machine code with respect to high-level specifications.
- We handle novel features of the execution model that are specific to its use in blockchain applications.
- We explain how we managed to carry out our work in an industrial setting, while the language and compiler were under continuous development.
- We explain why our approach, which involves automatically generating source-level proofs that are elaborated and checked by Lean, has been surprisingly effective.
- We demonstrate that our approach scales well by presenting a substantial case study, an implementation of digital signature validation that is already being used in production.

Our Lean libraries, our verification tool, and the case study described here can be found online at `https://github.com/starkware-libs/formal-proofs`. At the time of writing, version 0.10 of the Cairo language has been released, and the verifier and libraries accompanying this paper correspond to that release. StarkWare is currently developing the next generation of the programming language, Cairo 1, which will be substantially different. Cairo 1, however, will call libraries and primitives implemented in Cairo 0, and this project is currently being used to verify those libraries and primitives. All references to the Cairo language in this paper therefore refer to Cairo 0.

2 The Cairo Machine Model

The Cairo machine model is based on a simple CPU architecture with three registers: a *program counter* (pc), which points to the current instruction in memory; a *frame pointer* (fp), which generally points to the location of the local variables in a function call, and an *allocation pointer* (ap), which generally points to the next free value in global working memory. A machine instruction consists of three 16-bit words, which generally serve as memory offsets for operations performed on memory, and 15 1-bit flags, which determine the nature of the instruction. The architecture is described in detail in the Cairo whitepaper [14], and a Lean formalization thereof is described in [4].

A notable feature of the machine model is that elements of memory, as well as the contents of the registers, are elements of the field of integers modulo a certain prime number (by default, $2^{251} + 17 \cdot 2^{192} + 1$). The CPU can add and multiply values, but it cannot ask whether one value is greater than another. Cryptographic primitives, described in Section 5, can be used to assert that the contents of a memory location represent the cast of an integer in a certain range. The core library uses values that are checked to lie in the interval $[0, 2^{128})$.

Another notable feature of the machine model is that the memory is read-only. To establish a computational claim on blockchain, a prover makes public a partial assignment to memory that typically includes the program that is executed and the agreed-upon input. The prover also makes public the initial and final state of the registers and the number of steps in the computation. A certificate published on blockchain establishes, modulo common cryptographic assumptions, that the prover is in possession of a full assignment to memory extending the partial one such that the program runs to completion in the given number of steps. The code is then carefully designed to ensure that this implies the claim that is of interest to the verifier. For example, to establish that a calculation yields a claimed result, the prover and verifier agree on a Cairo program that carries out the calculation, asserts that it is equal to the claimed value, and fails otherwise. A certificate that the program terminates successfully establishes the computational claim.

Reading Cairo programs takes some getting used to. An instruction like `x = y + 5` is often thought of as an assignment of the value $y + 5$ to the memory location allocated to `x`, but it is really an assertion that the prover has assigned values to the memory so that the equation holds. It is an interesting feature of the model that a Cairo program can depend on values in memory that are assigned by the prover but not made public to the verifier. For example, a program can establish that a value `x` is a perfect square by asserting that it is equal to `y * y` for some `y`, without sharing the value of `y` with the verifier.

The Cairo CPU instruction set includes a call instruction, a return instruction, conditional and unconditional jumps, an instruction to advance the allocation pointer, and instructions that make arithmetic assertions about values stored in memory. The file `cpu.lean` consists of less than 200 lines of Lean definitions that provide a formal specification of the CPU and the next state relation. The next state depends on the contents of memory, `mem`, and the values of the CPU registers, `s`, but since the memory never changes, the next state relation `next_state mem s t` need only specify the successor state `t` of the registers. If the program counter points to an assert instruction that fails, there is no successor state. If the program counter points to an ill-formed instruction, the value of `t` is nondeterministic, so verifying that a Cairo program has the intended semantics generally requires establishing that a successful run of the program does not encounter such an instruction. By convention, programs halt with a jump instruction to the to itself, that is, an implicit infinite loop. The cryptographic proof published on blockchain establishes, with high probability, that the program has reached such a state.

Our project is designed to enable users to prove that the successful execution of a program guarantees that a property of interest holds. To that end, we define the following predicate:

```
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)
```

This says that any sequence `exec` of states that starts with σ, proceeds according to the machine semantics with respect to the memory assignment `mem`, and ends with a halting instruction eventually reaches a step `i` along the way such that the register state `exec i` satisfies `P`. Note that the predicate `P` can also make reference to the contents of memory, so we can express that at step `i` the memory location referenced by a certain register has a certain value, or that the value of a fixed memory location has a certain property. More precisely, the predicate `P` κ τ takes a numeric value κ as well as a register state τ, and the `ensures` predicate says that there is a value of κ less than or equal to `i` such that `P` κ (`exec i`) holds. We will explain the use of κ in Section 5.

3 From Assembly Code to Machine Code

The Cairo compiler translates code written in the Cairo language to instructions in the Cairo assembly language [14, Section 5], which are then translated to machine instructions. Assembly instructions can also be inserted directly into Cairo programs. The first step toward bridging the gap between the Cairo programming language and Cairo machine code is therefore to model the Cairo assembly language in Lean. The file `soundness/assembly.lean` in our project provides a description of Cairo machine instructions in terms of the offsets and flags, and it defines a translation from that representation to 63-bit machine code instructions. It also defines Lean notation that approximates Cairo assembly-language syntax. For example, here are three elementary mathematical Cairo functions from the Cairo common library:

```
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 (); }
```

The first confirms that the argument `a` is the cast of a nonnegative integer less than 2^{128}, by asserting that it is equal to the value of memory at the address `range_check_ptr`, which is assumed to point to a block of elements that have been verified to have this property. The second confirms that `a` is less than or equal to `b` by calling `assert_nn`(`b` − `a`), and the third function combines the previous two properties. The curly brackets mean that the argument `range_check_ptr` is passed to, updated by, and returned implicitly by these functions. We explain range checking in more detail in Section 5.

The Cairo compiler compiles these functions to assembly code and then to machine instructions. The assembly code corresponding to `assert_nn_le` looks as follows:

```
[ap] = [fp + (-5)]; ap++
[ap] = [fp + (-4)]; ap++
call rel -11
[ap] = [fp + (-4)]; ap++
[ap] = [fp + (-3)]; ap++
call rel -11
ret
```

The details are not important. The function calls are carried out by copying the arguments, including the implicit range check pointer, to the end of the global memory used so far. The arguments are referenced relative to the frame pointer, so [`fp` + (−5)] denotes the value in memory at the address `fp` − 5. The call instructions update the program counter and frame pointer so that execution continues in the subroutine, and the return at the end restores the frame pointer and updates the program counter to the next instruction in the calling routine.

Our tool generates a Lean description of this assembly code. The notation isn't pretty; we use tick marks and funny tokens to avoid conflicting with other tokens that may be in use. For example, the Lean description of the `assert_nn_le` assembly code is as follows:

```
def starkware.cairo.common.math.code_assert_nn_le : list F := [
  'assert_eq['dst[ap] === 'res['op1[fp+ -5]];ap++].to_nat,
  'assert_eq['dst[ap] === 'res['op1[fp+ -4]];ap++].to_nat,
  'call_rel['op1[imm]].to_nat, -11,
  '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 ]
```

As developers, we only had to read such code for debugging, and we found the notation convenient. Our Lean representation is adequate in the sense that the assembly instructions can be transformed to machine instructions, which can, in turn, be transformed to the 63-bit numeric representations which are then cast to the finite field `F`. Users can evaluate definitions like the one above in Lean and check that the resulting numeric values are the same ones produced by the Cairo compiler, and hence are the same ones used in the STARK certificate generated by the Cairo runner. Our soundness proofs start with the assumption that these values are stored in memory and that the program counter is set accordingly.

The file `soundness/assembly.lean` establishes a small-step semantics for reasoning about instructions at the assembly level. For example, variants of the Cairo `call` instruction allow specifying the address of the target in various ways, either as an absolute or relative address, which can in turn be given as an immediate value or read from memory with an offset from either the allocation pointer or frame pointer. The theorem describing the behavior of this instruction is as follows:

```
theorem next_state_call {F : Type*} [field F] (mem : F → F)
  (s t : register_state F) (op0 : op0_spec) (res : res_spec) (call_abs : bool) :
(call_instr call_abs res).next_state mem s t ↔
  (t.pc = jump_pc s call_abs (compute_res mem s (op0_spec.ap_plus 1) res) ∧
    t.ap = s.ap + 2 ∧
    t.fp = s.ap + 2 ∧
    mem (s.ap + 1) = bump_pc s res.to_op1.op1_imm ∧
    mem s.ap = s.fp)
```

Read this as follows: given that the CPU registers are in state `s` and given the contents of memory `mem`, the call instruction with boolean flag `call_abs` and operand specifications `op0` and `res` results in the new state `t`, where the program counter is updated as indicated, the relevant return address and the current frame pointer are stored in memory at the current allocation pointer, the allocation pointer is increased by two, and the frame pointer is increased by two. The details of the computations `jump_pc`, `compute_res`, and `bump_pc` are not important. What *is* important is that for concrete values of `op0`, `res`, and `call_abs`, Lean's tactics (a term rewriter, a numeric evaluator, etc.) are powerful enough to compute specific values and *prove* that the functions have those values. For example, if a specific call instruction decreases the program counter by 100, Lean can prove that the `next_state` relation holds for a suitable state `t` with `t.pc = s.pc - 100`. This allows us to reason about the behavior of a block of assembly code by stepping through each instruction in turn. The proof of the `next_state_call` and others like it are fiddly but straightforward: it is just a matter of unfolding the definitions of the assembly language instructions and then relating the resulting machine instructions to the semantics defined in `cpu.lean`.

4 From Cairo Code to Assembly Code

Consider the procedure `assert_nn_le`, which takes field elements `a` and `b` and asserts that they are casts of integers in a certain range such that the one corresponding to `a` is less than or equal to the one corresponding to `b`. More precisely, the desired specification is as follows:

```
def spec_assert_nn_le (mem : F → F) (κ : ℕ)
  (range_check_ptr a b ρ_range_check_ptr : F) : Prop :=
∃ m n : ℕ, m < rc_bound F ∧ n < rc_bound F ∧ a = ↑m ∧ b = ↑(m + n)
```

The argument `ρ_range_check_ptr` denotes the return value of `assert_nn_le`, which is implicit in the Cairo code. We often use unicode characters, which are allowed in Lean but not Cairo, to ensure that identifiers that we introduce in specifications and proofs do not clash with the identifiers that we take from Cairo. The up arrows denote casts to the field `F`. Here the value of `rc_bound F` is assumed to be 2^{128}, so m and n represent 128-bit unsigned integers. The autogenerated specification of `assert_nn_le` merely says that the Cairo code calls the two auxiliary functions `assert_nn` and `assert_le`:

```
def auto_spec_assert_nn_le (mem : F → F) (κ : ℕ)
    (range_check_ptr a b ρ_range_check_ptr : F) : Prop :=
∃ (κ₁ : ℕ) (range_check_ptr₁ : F),
  spec_assert_nn mem κ₁ range_check_ptr a range_check_ptr₁ ∧
∃ (κ₂ : ℕ) (range_check_ptr₂ : F),
  spec_assert_le mem κ₂ range_check_ptr₁ a b range_check_ptr₂ ∧
κ₁ + κ₂ + 7 ≤ κ ∧
ρ_range_check_ptr = range_check_ptr₂
```

The role of κ, κ_1, and κ_2 will be discussed in Section 5. Notice that the autogenerated specification for `assert_nn_le` refers to the *user* specifications of `assert_nn` and `assert_le` rather than the autogenerated ones. Interleaving the two types of specifications is crucial for handling recursion, since our autogenerated specification of a recursive function invokes the user specification to describe the effects of the recursive calls. We handle loops in a similar way. More importantly, our approach means that when users have to reason about the autogenerated specification, they can make use of their own specifications of the dependencies. This enables them to verify complex programs in a modular way.

With the autogenerated specification in hand, the user's task is to write their own specification of `spec_assert_nn_le` and prove that it follows from the autogenerated one. Our verification tool then uses that in the proof of the following theorem, which asserts that the machine code meets the user specification:

```
theorem auto_sound_assert_nn_le
  (range_check_ptr a b : F)
  (h_mem : mem_at mem code_assert_nn_le σ.pc)
  (h_mem_0 : mem_at mem code_assert_nn (σ.pc - 9))
  (h_mem_1 : mem_at mem code_assert_le (σ.pc - 5))
  (hin_range_check_ptr : range_check_ptr = mem (σ.fp - 5))
  (hin_a : a = mem (σ.fp - 4))
  (hin_b : b = mem (σ.fp - 3)) :
ensures mem σ (λ κ τ,
  τ.pc = mem (σ.fp - 1) ∧ τ.fp = mem (σ.fp - 2) ∧ τ.ap = σ.ap + 14 ∧
  ∃ μ ≤ κ,
    rc_ensures mem (rc_bound F) μ (mem (σ.fp - 5)) (mem (τ.ap - 1))
    (spec_assert_nn_le mem κ range_check_ptr a b (mem (τ.ap - 1))))
```

The theorem asserts that, given the contents of memory `mem` and the register state σ, if the code for `assert_nn_le` is in memory at the program counter, the code for the dependencies are in place as well, and the arguments to the function are stored in memory in the expected locations indexed by the frame pointer, then any halting computation eventually returns to the calling function (restoring the program counter and frame pointer according to the Cairo language calling conventions) and ensures that the user specification holds, assuming that certain auxiliary locations in memory have been range checked. Once again, we promise to explain range checking in Section 5.

The proof of `auto_sound_assert_nn_le` establishes the correctness of the autogenerated specification and then applies the user-supplied theorem that this implies the user's specification. Generally speaking, the user doesn't need to see the Lean description of the assembly code, the theorem `auto_sound_assert_nn_le`, or the proof of correctness. The autogenerated specifications, the user specifications, and the proof that the former imply the latter are stored in the same directory as the Cairo code. The Lean descriptions of the assembly code and the correctness proofs are kept in a separate folder, tucked out of sight.

Our verification tool has the task of extracting the autogenerated specifications and constructing the correctness proofs. The Cairo compiler, which is written in Python, produces a number of data structures that we are able to make use of once the compilation is complete. These contain, for example, a dictionary of namespaced identifiers. Whenever we needed additional information, we added hooks that, in verification mode, are called to log that information. For example, a compound assertion like $\texttt{x} = 3 * \texttt{y} + 4 * \texttt{z}$ translates to a list of atomic assertions, and our verification tool has access to the original equation and the code points that mark the beginning and end of the list of assertions.

As we will discuss in Section 5, the Cairo language uses two sorts of variables: local variables are indexed with offset from the frame pointer, and global variables are indexed offset from the allocation pointer. When a function takes values `a b c : F` as arguments, the compiler places these values in memory just before the allocation pointer when the procedure is called. For the most part, the verifier keeps the nitty-gritty memory allocation issues hidden from the user, and mediates between variable names and the machine semantics with with equations like $\texttt{a} = \texttt{mem}\ (\sigma.\texttt{ap} - 3)$. The Cairo language also allows the definition of compound structures, and we define the corresponding structures in Lean and interpret references to memory accordingly.

Our verifier uses Dijkstra's weakest preconditions [10] to read off a specification. The process is straightforward, modulo the fact that the verifier also has to construct Lean proofs that prove that these specifications are met. That requires unpacking the meaning of each machine instruction, using the theorems described in the previous section. Unpacking the `mem_at` predicate tells us which instruction is present at each memory location. We then use special-purpose tactics (small-scale automation written in Lean) to unpack the effect of each instruction. For example, suppose at a given point in the proof we need to show that executing the code at program counter $\sigma.\texttt{pc} + 5$ ensures that a certain result holds, and we know that the instruction at that location corresponds to a certain instruction with an immediate value. Applying the relevant tactic leaves us the goal of showing that executing the code at program counter $\sigma.\texttt{pc} + 7$ ensures the desired result, with the other registers and the proof context updated to reflect the result of executing the instruction. This reduction is justified by appealing to the meaning of the `ensures` predicate and the specification of the machine semantics. In this way, our tactics carry out a kind of symbolic execution of the assembly code, and register the effects in the proof context.

The verifier's task is then to parse each high-level Cairo instruction, generate a specification of its behavior, and construct the corresponding part of the correctness proof, which shows that the corresponding assembly instructions implement the high-level ones.

- A variable declaration like tempvar $b = a + 5$ translates to an existential quantifier in the specification, $\exists$ b, $b = a + 5 \wedge \ldots$. The correctness proof instantiates the existential quantifier to the corresponding memory location and maintains this correspondence.
- An equality assertion in the program translates to an equality assertion in the specification. Such an assertion generally translates to one or more assembly-level assertions, and the correctness proof involves reconstructing the compound equality statement from the components.
- Cairo programs can have labels and both conditional and unconditional jumps. The corresponding machine code has the expected effect of (conditionally) modifying the program counter. In some settings, the correctness proof only needs to record this change to the program counter and continue stepping through the instructions starting at the new pc. But for handling jumps that coalesce control flow, and loops in particular, we analyze the control flow into blocks and break the specification and correctness proof up accordingly. We describe this process further below.
- A conditional jump is implicit in a Cairo if ... then ... else construct. This translates to a disjunction in the specification and the definition of a block where the branches flow together.
- A subroutine call to another procedure translates to an assertion, in the autogenerated specification, that the user specification of the target procedure holds of the arguments and the return value. The call instruction stores the current program pointer and frame pointer in memory and jumps to the location of the target procedure. The return instruction restores the frame pointer and jumps back to the calling procedure. The correctness proof invokes the correctness theorem for the target procedure as well as the assumption that the procedure is in memory at the expected location.

The description so far presupposes that the control flow has no cycles. To handle recursive calls and loops, we do not have to prove termination; the STARK certificate assures a skeptical verifier that the program has terminated, so we need only show that, given that fact, the specification is met. (This is commonly characterized as the difference between *partial correctness* and *total correctness*.) The claim `ensures mem` σ `P` is equivalent to $\forall$ `b`, `ensuresb b mem` σ `P`, where `ensuresb b mem` σ `P` says that every halting execution sequence from state σ with at most `b` steps eventually reaches a state that satisfies `P`. We can prove the latter by induction on `b`, generalizing over states σ with program counter pointing to the relevant code.

For functions that call themselves recursively, we modify the default user specification so that it is trivially true, and place it *before* the autogenerated specification. The autogenerated specification asserts the play-by-play description alluded to above, except that it uses the user specification to characterize the recursive calls. The user is free to write any specification they want, provided they show that the autogenerated specification implies the user specification. In short, the user has to show that the play-by-play characterization implies their own characterization, assuming their characterization holds at downstream calls. The correctness proof uses this together with the inductive hypothesis at downstream calls.

A similar method handles loops. Our verification tool begins by analyzing the control-flow graph [2], dividing the code into basic blocks, without any jumps or labels. A block starts at the beginning of a function or at a label, and ends with a jump, a return, or a flow to a label. Any block that can be entered from more than one other block receives a separate specification in the specification file, and cycles that arise in a topological sort are handled in a manner similar to recursive function calls. In practice, the user can specify that the execution has an effect conditional on an invariant holding at the entry point. Verification of the full user specification then requires showing that the invariant holds at the first entry point and that it is maintained on re-entry.

5 Memory Management and Range Checks

In this section, we discuss aspects of the Cairo programming language that stem specifically from its intended use toward verifying computations on blockchain. Encoding execution traces efficiently required keeping the machine model simple, which is why StarkWare's engineers settled on a CPU with only three registers and read-only memory. The cost of certification on blockchain scales with the number of steps in the execution trace together with the number of memory accesses. The fact that memory is read-only makes the verification task easier: we do not need to worry about processes overwriting each other's memory. The Cairo language allows procedures to declare two types of temporary variables, namely, relative to the frame pointer (fp) or allocation pointer (ap). It is a quirk of the Cairo language that references to ap-based variables can be revoked when the compiler cannot reliably track the effects of intermediate commands and function calls on the ap, for example, when different flows of control may result in different changes to the ap. Our verification relies on the compiler's internal record of its ability to track these changes.

The STARK encoding is most efficient when memory is assigned in a continuous block. The Cairo compiler is tightly coupled with a Cairo *runner*, whose task is to allocate memory and assign values to ensure that the Cairo program runs to completion. The Cairo whitepaper describes the methods that are used to simulate conventional memory models. The processor uses the frame pointer to point to the base of a procedure's local memory and the allocation pointer to point to the next available position in global memory. Local variables are kept in the same contiguous block. When one procedure calls another, the program counter and frame pointer are stored in global memory, the allocation pointer is updated, and the frame pointer is set equal to the allocation pointer. When the procedure returns, the program counter and frame pointer are restored.

For efficiency, a local procedure sometimes has to access values that are stored in global memory, which is to say, they are indexed relative to the allocation pointer. This is challenging because the allocation pointer is constantly changing. For example, when one procedure calls another, upon the return the allocation pointer may have changed. Moreover, the new value of the allocation pointer cannot always be predicted at compile time; for example, different flows of control through an if-then-else can result in different changes to its value. The compiler uses a *flow tracker* that keeps track of these changes as best it can, allowing the programmer to refer to the same global variables throughout. Our verification tool does not have to know much about how the flow tracker works, but it needs to make use of the results. For example, if the value of a variable `x` is `mem` $(\texttt{ap} + 1)$ before a subroutine call and the allocation pointer `ap'` on return is equal to $\texttt{ap} + 3$, our Lean proofs need to use the identity $\texttt{ap'} = \texttt{ap} + 3$ to translate an assertion involving `mem` $(\texttt{ap'} - 2)$ into an assertion about `x`. Our verification tool claims and proves the relevant identities while stepping through the code, and uses those identities as rewriting rules when verifying assertions.

A more striking difference between programming in Cairo and programming in an ordinary programming language is that the most fundamental data type consists of values of a finite field. One can add and multiply field elements, but there is no machine instruction that compares the order of two elements. To meet high-level specifications that are stated in terms of integers, the STARK encoding uses cryptographic primitives to verify that a specified range of memory has been *range checked*, which is to say, the corresponding field elements are casts of integers in the interval $[0, 2^{128})$. Our previous verification of the STARK encoding [4] shows that the STARK certificate guarantees (with high probability) that the specified memory locations have indeed been range checked. A Cairo program can make use of this fact by taking, as input, a pointer to a location in the block of range-checked memory, making assertions about a sequence of values at that location, and returning (in addition to its ordinary return values) an updated pointer to the next unused element. Both the user specification and the autogenerated specification are of the form "assuming the values between ... and ... have been range-checked, the following holds: ... ." These hypotheses have to be used inside the correctness proofs, to justify the assertions that particular values have been range checked. The hypotheses also have to be threaded through procedure calls and combined appropriately, so the specification of a top-level function comes with a range-check hypothesis that covers all the recursive calls. This top-level hypothesis is justified by the STARK certificate.

Our verification tool handles all this plumbing. For example, recall the Cairo function `assert_nn`, which asserts that the argument `a` is the cast of an integer in $[0, 2^{128})$.

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

The curly brackets in `{range_check_ptr}` indicate that the value should implicitly be returned among the other return values. (In this case, there aren't any others.) The user-written specification of this function is as follows:

```
def spec_assert_nn (mem : F → F) (κ : ℕ)
    (range_check_ptr a ρ_range_check_ptr : F) : Prop :=
  ∃ n : ℕ, n < rc_bound F ∧ a = ↑n
```

Recall that the annotation `↑n` casts the natural number `n` to the underlying field `F`. Our verification tool generates the following specification:

```
def auto_spec_assert_nn (mem : F → F) (κ : ℕ)
    (range_check_ptr a ρ_range_check_ptr : F) : Prop :=
  a = mem (range_check_ptr) ∧
  is_range_checked (rc_bound F) a ∧
  ∃ range_check_ptr₁ : F, range_check_ptr₁ = range_check_ptr + 1 ∧
  3 ≤ κ ∧
  ρ_range_check_ptr = range_check_ptr₁
```

Here, $\texttt{range\_check\_ptr}_1$ is the updated version of `range_check_ptr`, and the last line specifies that this is the function's sole return value. Our tool detects the reference to `range_check_ptr` in the Cairo code and adds `is_range_checked` (`rc_bound F`) `a` to the autogenerated specification, generating an obligation in the correctness proof that the user does not have to see. The user has to prove that `spec_assert_nn` follows from `auto_spec_assert_nn`, but this follows immediately from the conjunct `is_range_checked` (`rc_bound F`) `a` in the autogenerated specification.

The ultimate correctness theorem is stated as follows:

```
theorem auto_sound_assert_nn
  (range_check_ptr a : F)
  (h_mem : mem_at mem code_assert_nn σ.pc)
  (hin_range_check_ptr : range_check_ptr = mem (σ.fp - 4))
  (hin_a : a = mem (σ.fp - 3)) :
ensures mem σ (λ κ τ,
  τ.pc = mem (σ.fp - 1) ∧ τ.fp = mem (σ.fp - 2) ∧ τ.ap = σ.ap + 1 ∧
  ∃ μ ≤ κ, rc_ensures mem (rc_bound F) μ (mem (σ.fp - 4)) (mem (τ.ap - 1))
    (spec_assert_nn mem κ range_check_ptr a (mem (τ.ap - 1))))
```

In this specification, the assertions $\tau.\texttt{pc} = \texttt{mem}\ (\sigma.\texttt{fp} - 1)$ and $\tau.\texttt{fp} = \texttt{mem}\ (\sigma.\texttt{fp} - 2)$ assert that the program counter and frame pointer have been restored correctly when the function returns. Our verification tool learns from the flow tracker that any path through this code updates the allocation pointer by one, and so it also establishes that fact, i.e. $\tau.\texttt{ap} = \sigma.\texttt{ap} + 1$, to make that information accessible when reasoning about procedures that call it. The `rc_ensures` clause in the conclusion says that if the block of memory between $\texttt{mem}\ (\sigma.\texttt{fp} - 4)$ and $\texttt{mem}\ (\tau.\texttt{ap} - 1)$ is range-checked then the user specification holds. (We will return to the role of μ in a moment.) Here $\sigma.\texttt{fp}$ is the value of the frame pointer when the function is called, $\tau.\texttt{ap}$ refers to the value of the allocation pointer upon return, and $\texttt{mem}\ (\sigma.\texttt{fp} - 4)$ and $\texttt{mem}\ (\tau.\texttt{ap} - 1)$ are, respectively, the location of the argument `range_check_ptr` and the return value, which is supposed to be the updated range check pointer.

We can now explain the role of κ and μ. Recall that $\texttt{mem}\ (\sigma.\texttt{fp} - 4)$ and $\texttt{mem}\ (\tau.\texttt{ap} - 1)$ are field elements. A first guess as to how to specify that the range of memory values between those two locations is range checked is to say that there is a natural number μ such that $\texttt{mem}\ (\tau.\texttt{ap} - 1) = \texttt{mem}\ (\sigma.\texttt{fp} - 4) + \uparrow\mu$ and for every $\texttt{i} < \mu$, the value in memory at address $\texttt{mem}\ (\tau.\texttt{ap} - 1) + \texttt{i}$ is range checked. But this specification is problematic: if the equation holds for some small value of μ, it also holds for μ plus the characteristic of the underlying field. Our correctness proof needs to use the fact that the total number of range-checked elements does not wrap around the finite field. We achieve this by asserting that μ is, moreover, bounded by the number of steps κ in the execution trace, which is made public in the STARK certification and is always smaller than the characteristic of the field. In the case of range checks, the bounds are handled entirely by the verifier and the user need not worry about them. But we have found that some Cairo specifications require similar reasoning about bounds on the length of the execution, and for those rare occasions, we have exposed the parameter κ in the user-facing specifications.

The virtue of our verification tool is that the user can be oblivious to most of the implementation details we have just described, such as the handling of the range check pointers and the way that variables, arguments, and return values are stored in memory. The user writes the Cairo procedure `assert_nn` and is given the specification `auto_spec_assert_nn`. The user then writes the specification `spec_assert_nn` and proves that `spec_assert_nn` follows from `auto_spec_assert_nn`. The correctness proof can be checked behind the scenes. From that moment on, `spec_assert_nn` is all that users need to know about the behavior of `assert_nn`, from the point of view of proving properties of Cairo functions that use it. In the next section, we will show that this scales to the verification of more complex programs.

6 Validating Digital Signatures

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*. The set of such points has the structure of an abelian group where the zero is defined to be the point at 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 $(x_0, y_0) + (x_1, y_1) = (x', y')$.

It is not hard to prove that with addition, negation, and zero so defined, the structure satisfies all the axioms for an abelian group other than associativity. Proving associativity is trickier, though it can be done with brute-force algebraic computations in computer algebra systems, and various approaches have been used in the interactive theorem proving literature to establish the result formally [25, 5, 12, 15, 3].

The study of elliptic curves over the complex numbers originated in the nineteenth century, where the addition law has a geometric interpretation. The topic is fundamental to contemporary number theory. Elliptic curves over a *finite field* are widely used in cryptography today, on the grounds that for any nonzero point x, the map $n \mapsto n \cdot x$ (that is, n-fold sum of x with itself) is easy to compute but, as far as we know, difficult to invert. This forms the basis for the *elliptic curve digital signature algorithm* (ECDSA). ECDSA provides a protocol by which a sender can generate a pair consisting of a public key and a private key, publish the public key, and then send messages in such a way that a receiver can verify that the message was sent by the holder of the private key and that the message has not been changed.

The Cairo library contains functions that support ECDSA over the secp256k1 elliptic curve, that is, the curve $y^2 = x^3 + 7$ over the finite field of integers modulo the prime $p = 2^{256} - 2^{32} - 977$. For reasons we will shortly explain, the calculations are subtle. We have proved the correctness of the Cairo functions implementing the elliptic curve operations efficiently, as well as a Cairo procedure for validating secp signatures. Figure 1 shows the Cairo procedure for recovering the public key of the sender from a digitally signed message, and Figure 2 shows the correctness proof that we have obtained in Lean. The rest of this section is devoted to describing the formalization and the resulting theorem.

```
func recover_public_key{range_check_ptr}(msg_hash: BigInt3,
    r: BigInt3, s: BigInt3, v: felt) -> (public_key_point: EcPoint) {
  alloc_locals;
  let (local r_point: EcPoint) = get_point_from_x(x=r, v=v);
  let (generator_point: EcPoint) = get_generator_point();
  let (u1: BigInt3) = div_mod_n(msg_hash, r);
  let (u2: BigInt3) = div_mod_n(s, r);
  let (point1) = ec_mul(generator_point, u1);
  let (minus_point1) = ec_negate(point1);
  let (point2) = ec_mul(r_point, u2);
  let (public_key_point) = ec_add(minus_point1, point2);
  return (public_key_point); }
```

Figure 1 Cairo procedure for recovering an secp public key.

```
def spec_recover_public_key (mem : F → F) (κ : ℕ)
  (range_check_ptr : F) (msg_hash r s : BigInt3 F)
  (v ρ_range_check_ptr : F) (ρ_public_key_point : EcPoint F) : Prop :=
∀ (secpF : Type) [secp_field secpF], by exactI
r ≠ ⟨0, 0, 0⟩ →
∀ ir : bigint3, ir.bounded (3 * BASE - 1) → r = ir.toBigInt3 →
∀ is : bigint3, is.bounded (3 * BASE - 1) → s = is.toBigInt3 →
∀ imsg : bigint3, imsg.bounded (3 * BASE - 1) → msg_hash = imsg.toBigInt3 →
∃ nv : ℕ, nv < rc_bound F ∧ v = ↑nv ∧
∃ iu1 iu2 : ℤ,
  iu1 * ir.val ≡ imsg.val [ZMOD secp_n] ∧
  iu2 * ir.val ≡ is.val [ZMOD secp_n] ∧
∃ ny : ℕ, ny < SECP_PRIME ∧ nv ≡ ny [MOD 2] ∧
∃ h_on_ec : @on_ec secpF _ (ir.val, ny),
∃ hpoint  : BddECPointData secpF ρ_public_key_point,
  hpoint.toECPoint =
    -(iu1 · (gen_point_data F secpF).toECPoint) +
      iu2 · ECPoint.AffinePoint ⟨ir.val, ny, h_on_ec⟩
```

Figure 2 Correctness of the digital signature validation.

To start with, in the file `elliptic_curves.lean`, we define the secp curve over an arbitrary finite field of odd characteristic, define the group operations, and prove that they form a group, modulo a proof that the group law is associative. Lean allows us to insert a `sorry` placeholder for the missing proof of associativity; this is the only `sorry` in our development. David Kurniadi Angdinata and Junyan Xu have recently verified, in Lean, that the elliptic curve law forms a group, in impressive generality [3]. This will allow us to eliminate the `sorry`.

The files `constants.cairo`, `bigint.cairo`, `field.cairo`, and `ec.cairo` implement the operations over the secp curve, culminating in an efficient procedure to carry out scalar multiplication. The main reason that the code is subtle is that it requires calculations in the field $\mathbb{Z}/p\mathbb{Z}$, where p is the secp prime, which is even larger than (and different from!) the characteristic of the field that underlies the Cairo machine model. The Cairo implementation thus represents a value x in $\mathbb{Z}/p\mathbb{Z}$ by three field elements, each of which is checked to be the cast of an integer in a certain range. We impose additional bounds and hypotheses on these representations, and ensure that they are maintained by the calculations.

In greater detail, the Cairo code defines a constant `BASE` equal to 2^{86} and a structure `BigInt3 {d0: felt, d1: felt, d2: felt}`, with the intention that the field elements `d0`, `d1`, and `d2` will always be casts of integers `i0`, `i1`, and `i2`, respectively, with absolute values in $[0, 3 \cdot \texttt{BASE})$. These are intended to represent the value $\texttt{i0} + \texttt{i1} \cdot \texttt{BASE} + \texttt{i2} \cdot \texttt{BASE}^2$. Note that the values `i0`, `i1`, and `i2`, may be larger than `BASE`, so these representations are not unique. Our specification files define a Lean structure `bigint3` := (`i0 i1 i2` : ℤ), as well as a predicate `bigint3.bounded i b` that says that each of the three limbs of the `bigint3` denoted by `i` is bounded in absolute value by `b`. 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 `F` of the Cairo machine model, assumed or checked to be casts of such integers.

Field operations like addition and multiplication on the secp field correspond to addition and multiplication on the integer representations modulo the secp prime. These in turn are carried out by Cairo code on the triples of field elements, with care to ensure that the results track the corresponding operations on suitable integer representations.

An element of the secp curve consists of a pair (x, y) of elements of the secp field satisfying $y^2 = x^3 + 7$ or the special point at infinity. These are represented in the Cairo code by a structure `EcPoint` given by `x: BigInt3, y: BigInt3`, with the point at infinity represented by any pair with `x` equal to the triple ⟨0, 0, 0⟩. (This works because 7 is not a square modulo the secp prime.) We therefore use the following data structure to express that `pt : EcPoint` represents a point on the curve.

```
structure BddECPointData (secpF : Type*) [field secpF] (pt : EcPoint F) :=
  (ix iy : bigint3)
  (ixbdd : ix.bounded (3 * BASE - 1))
  (iybdd : iy.bounded (3 * BASE - 1))
  (ptxeq : pt.x = ix.toBigInt3)
  (ptyeq : pt.y = iy.toBigInt3)
  (onEC : pt.x = ⟨0, 0, 0⟩ ∨ (iy.val : secpF)^2 = (ix.val : secpF)^3 + 7)
```

The specification of a procedure that takes an `EcPoint` as input generally also assumes that the `EcPoint` is equipped with such data, and the specification of a procedure that *ouputs* an `EcPoint` generally *proves* the existence of the corresponding data.

We can now explain the Cairo code in Figure 1 and the specification in Figure 2. The digital signature method used by Cairo requires that the sender and receiver agree on the elliptic curve they are using and on a message hash function. They also fix a point G on the curve that generates the group, which has a known prime order n. The sender applies a hash function to the message to obtain an integer m, and the method provides a recipe for the sender to generate a triple (r, s, v) where r and s are integers and v is an additional bit. The recipient of the message and the signature applies the hash function to obtain m as well, checks to make sure $r \neq 0$, then finds a point (r, y) on the elliptic curve by finding the residues y satisfying $y^2 = r^3 + 7$ in the secp field and choosing the one that has the same parity as v. The receiver then computes $u_1 = r^{-1} \cdot m$ and $u_2 = r^{-1} \cdot s$, where these operations take place in the group of residues modulo n. Using scalar multiplication, the receiver calculates $Q = -u_1 \cdot G + u_2 \cdot (r, y)$, a point on the elliptic curve. If the value Q matches the sender's public key, the receiver has the desired confirmation that the message m has been sent by the sender.

The procedure `recover_public_key` carries out exactly the calculation of Q. It takes as input elements `msg_hash`, `r`, `s`, and `v` in the Cairo field. In the specification, the first three are assumed to be casts of suitably bounded integers `imsg`, `ir`, and `is`. In other words, these assumptions should be guaranteed by the calling procedure. It then checks that `v` is the cast of a suitably bounded natural number `nv` (this representation is necessarily unique), and it confirms the existence of data `hpoint` representing the point $-u_1 \cdot G + u_2 \cdot (r, y)$ in the calculation above.

The implementation of the procedure requires subtle calculations and checks. The best explanation of what the intermediate calculations are supposed to achieve are given by the Lean specification files themselves. The formalization uses the library file `math.cairo`, which runs about 450 lines of code; our Lean specifications of those functions, as well as our proofs of our own specifications from the autogenerated ones, comprises about 1,150 lines of code. The secp validation procedure runs about 800 lines of Cairo code and our specification files run about 3,200 lines of Lean code, on top of about 150 lines in our definition of the elliptic curve group. The dependency chain of `recover_public_key` consists of 24 Cairo functions, which compile to about 900 lines of assembly code, i.e. 900 field elements. Our autogenerated correctness proofs run about 7,500 lines of Lean code.

Verifying an early version of the secp code turned up two errors that were independently caught and fixed by the software engineers. The verification later turned up an error that they missed, having to do with the use of the parameter `v` in `recover_public_key`. The error, which does not allow the prover to fake a signature but does allow it to claim that a valid signature is invalid, was fixed in the next Cairo release. Beyond that, the verification provided the software engineers with welcome reassurance. Despite extensive code review, they recognized that there were a number of places where small errors may have crept into the code, and they were able to breathe a sigh of relief when the verification was complete.

7 Methodology

We have reported on the means we have developed to deal with quirks of the Cairo architecture that stem from the need to encode Cairo computations efficiently in a STARK certificate. Beyond that, many of the methods we have used are routine for software verification. But some aspects of the way we have implemented these methods are notable, since they have enabled us to put the methods to use in a production setting. In this section, we discuss some of the pragmatic choices we have made and assess their effectiveness.

It is notable that our end-to-end correctness proofs are carried out within a single foundational proof assistant. Systems such as Why3 [13], Dafny [18], and F^* [24] extract verification conditions from imperative programs, but they do not generally verify those conditions with respect to a mathematical specification of a machine model. These systems also tend to rely on automation, like SMT solvers, that has to be trusted. In contrast, all our theorems are stated in the context of a precise axiomatic foundation and the proofs are checked by a small trusted kernel, for which independent reference checkers are available. This provides a high degree of confidence that the machine code meets its high-level specifications. Similar approaches to verifying code with respect to machine semantics include MM0 [7] and [21].

Another advantage of embedding the verification in a foundational proof assistant is that the availability of an ambient mathematical library [20] means that we can make use of any mathematical concepts that are needed to make sense of the high-level specification. Our verification of the digital signature recovery algorithm required reasoning about elliptic curves, as well as dealing with bounds and casts of integers to a finite field. We were able to carry out this reasoning in the same proof assistant that we used to carry out low-level reasoning about the machine code.

It is notable that the development of our tooling did not hamper the development of the Cairo compiler or its library. When we began our project, the compiler was already being used in production, and it is still under continuous development. Requesting substantial changes to the compiler code base would have slowed our efforts, requiring not only coordination with the compiler team but also extensive code review. With our approach, we were able to work under the radar, harvesting just enough data from the compiler for us to construct our proofs. For example, we found that justifying equality assertions between compound terms did not require a detailed understanding of the process by which the compiler carried out the calculations; it was enough to simply keep track of the intermediate assertions and pass those equations to Lean's simplifier.

An alternative approach to end-to-end verification is to verify a compiler with respect to a deeply embedded semantics. This is the approach taken by CompCert [19], CakeML [17, 1], and the Bedrock project (e.g. [8]). But the Cairo language is still evolving and there is no formal specification of its semantics, even though the meaning of a Cairo program is intuitively clear in general. Our approach gives us the freedom to generate specifications with confidence that they are correct, since they are backed up by formal proof. Producing proofs of correctness at compile time avoids having to model parts of the compiler that are irrelevant to correctness, and it does not require us to find a clean separation between those parts and the ones that are. It also avoids the need to verify behaviors that don't arise in practice. For example, Cairo allows for arbitrary labels and jumps, and programmers are free to write whatever spaghetti code they want. Our verification tooling is designed to work on regular control flow graphs, and will simply fail otherwise. This leaves the decision with Cairo developers as to whether to revise their Cairo code to fit our verification model, to verify their code by hand, or to leave it formally unverified. Thus our approach provides tools that are effective in practice without dictating or constraining the language development.

Perhaps most striking is our decision to construct correctness proofs by generating Lean source code that is then elaborated and checked by the same Lean process that elaborates and checks hand-written proofs. This means that our tool automatically constructs long, complex proofs in a system that has been carefully designed to support synergetic user interaction. This may seem odd and counterproductive. But we found that the compilation process is deterministic enough to make it possible to construct these proofs, and that we could make use of similarly deterministic and predictable automation in Lean.

Moreover, we found that the approach supports an efficient workflow for the developers of the verification tool as well as for users of the tool who wish to verify their Cairo specifications. To verify a Cairo program, a user runs our verification tool on the main file, which can import other Cairo program files. Our tool calls the compiler to compile the program and then generates a Lean description of the compiled code, a Lean specification file for each Cairo source file, and proofs that the compiled code meets the specifications. By convention, the specification files, which are typically the only formal content the user needs to inspect and modify, are kept with the source files. For example, a Cairo file `foo.cairo` gives rise to a specification file `foo_spec.lean` in the same directory. The remaining files are kept in a `verification` folder in the directory containing the main Cairo source file. Compiling the files immediately after the tool is run confirms that the compiled code meets the autogenerated default specifications. Compiling them again after the user adds their own specifications to the spec files and proves that they follow from the autogenerated specifications ensures that the code meets their specifications. The correctness proofs do nothing more than apply the theorem that says that the autogenerated specification implies the user one, so if the initial correctness proofs have already been checked and Lean accepts the proofs in the user specification file, it is unlikely that the subsequent check of the correctness proofs will fail. For the application described in Section 6, compiling and checking all the files in Lean requires only a few minutes on an ordinary desktop. Compiling and checking the specification files alone, with the user specifications and correctness proofs, takes less than two minutes from scratch. In practice, the user files are checked incrementally in real time, as the user types them into the editor.

This results in a congenial workflow. After first running the verification tool, a user can compile the files in the verification folder to confirm that the specifications are well formed and the correctness proofs are valid. The user can then focus on writing the specifications and proving them correct. We have arranged it so that if the verification tool is run again in the presence of existing specification files, we do not overwrite any of the user-supplied content. We only add or change the autogenerated specifications, as well as the arguments to the specifications when the arguments to the corresponding Cairo functions change. (The tool leaves comments in the file so that the user can see what has changed.) That way, when the Cairo code changes, the user only needs to make corresponding changes to the specifications. Moreover, when verifying another Cairo file with overlapping dependencies, one can make use of the same specifications. This has made it possible for us to verify the Cairo library one step at a time.

Our approach has also had important benefits for the development of the verification tool. We started our project by compiling simple programs, extracting Lean descriptions of the compiled code, and writing and proving specifications by hand. This helped us determine what the autogenerated specifications should look like and taught us how to construct the correctness proofs. We then simply had to write Python code that did the same thing automatically. We were able to iteratively extend the tool to handle other aspects of the Cairo language: if-then-else blocks, recursive calls, structures, loops, and so on. As we worked through files in the Cairo library, whenever we came across a feature the verifier was not equipped to handle, we could figure out how to handle the feature manually, and then extend the tool to handle that and future instances. Debugging was similarly straightforward: whenever one of our autogenerated proofs failed, we could open the file, go to the error, and use Lean's rich editor interface to inspect the proof state. Once we figured out how to repair the error manually, it was generally not hard to modify the verification tool to produce the desired behavior automatically. Our generated code is structured, commented, and readable. It slightly more verbose and formulaic than proofs one would write by hand, but it is otherwise similar.

In sum, formal verification requires a synergetic combination of automation and user interaction. One of our most important findings is that using automation to generate formal content that can be inspected and modified interactively is a remarkably powerful and effective means to that end.

8 Conclusions and Related Work

We have presented a means of verifying functional correctness of programs written in the Cairo language with respect to a low-level machine model, and we have demonstrated its practical use with a case study, in which we have verified a Cairo library procedure for validating cryptographic signatures. We have similarly verified other fundamental components of the Cairo library, including a procedure that Cairo programmers can use to simulate the behavior of read-write dictionaries in Cairo's read-only memory model [14, Section 8.5.2].

In Section 7, we have already cited some other approaches toward verifying a functional specification down to machine code, and in Section 6, we cited various formalizations of the associativity of the group law for elliptic curves. In recent years, there has been extensive work on verification of cryptographic primitives [11, 22, 23, 26], including the kind of digital signature recovery described here. As we have explained, however, verification of Cairo programs requires dealing with specific features of the language and machine model, and it is notable that we have achieved end-to-end verification in a foundational proof assistant. Because Cairo programs are used extensively to carry out financial transactions, and because they are carefully optimized to reduce the cost of on-chain verification, having workable means of verifying their correctness is essential.

References

1 Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. Proof-producing synthesis of CakeML from monadic HOL functions. *J. Autom. Reason.*, 64(7):1287–1306, 2020. doi:10.1007/s10817-020-09559-8.

2 Frances E. Allen. Control flow analysis. *SIGPLAN Notices.*, 5(7):1–19, July 1970. doi:10.1145/390013.808479.

3 David Kurniadi Angdinata and Junyan Xu. An elementary formal proof of the group law on weierstrass elliptic curves in any characteristic. In Adam Naumowicz and René Thiemann, editors, *Interactive Theorem Proving (ITP) 2023*, 2023.

4 Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman. A verified algebraic representation of Cairo program execution. In Andrei Popescu and Steve Zdancewic, editors, *Certified Programs and Proofs (CPP) 2022*, pages 153–165. ACM, 2022. doi:10.1145/3497775.3503675.

5 Evmorfia-Iro Bartzia and Pierre-Yves Strub. A formal library for elliptic curves in the Coq proof assistant. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP) 2014*, pages 77–92. Springer, 2014. doi:10.1007/978-3-319-08970-6_6.

6 Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018:46, 2018. URL: http://eprint.iacr.org/2018/046.

7 Mario Carneiro. Metamath Zero: Designing a theorem prover prover. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics (CICM) 2020*, pages 71–88. Springer, 2020. doi:10.1007/978-3-030-53518-6_5.

8 Adam Chlipala. The Bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In Greg Morrisett and Tarmo Uustalu, editors, *International Conference on Functional Programming (ICFP) 2013*, pages 391–402. ACM, 2013. doi:10.1145/2500365.2500592.

9 Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Conference on Automated Deduction (CADE) 2015*, pages 378–388. Springer, Berlin, 2015. doi:10.1007/978-3-319-21401-6_26.

10 Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. doi:10.1145/360933.360975.

11 Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic: With proofs, without compromises. *ACM SIGOPS Oper. Syst. Rev.*, 54(1):23–30, 2020. doi:10.1145/3421473.3421477.

12 Andrew Erbsen. *Crafting Certified Elliptic Curve Cryptography Implementations in Coq*. PhD thesis, Massachusetts Institute of Technology, 2017.

13 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — Where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *European Symposium on Programming (ESOP) 2013: Programming Languages and Systems*, pages 125–128. Springer, 2013. doi:10.1007/978-3-642-37036-6_8.

14 Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo — a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Report 2021/1063, 2021. URL: https://ia.cr/2021/1063.

15 Thomas C. Hales and Rodrigo Raya. Formal proof of the group law for edwards elliptic curves. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *International Joint Conference on Automated Reasoning (IJCAR) 2020*, pages 254–269. Springer, 2020. doi:10.1007/978-3-030-51054-1_15.

16 StarkWare Industries. Cairo. https://www.cairo-lang.org/.

17 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *Principles of*

Programming Languages (POPL) 2014, pages 179–192. ACM, 2014. doi:10.1145/2535838.2535841.

18 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) 2010- 16th*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.

19 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

20 The mathlib community. The Lean mathematical library. In Jasmin Blanchette and Catalin Hritcu, editors, *Certified Programs and Proofs (CPP) 2020*, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.

21 Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Extensible proof-producing compilation. In Oege de Moor and Michael I. Schwartzbach, editors, *Compiler Construction (CC) 2009*, pages 2–16. Springer, 2009. doi:10.1007/978-3-642-00722-4_2.

22 Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *IEEE Symposium on Security and Privacy (SP) 2020*, pages 983–1002. IEEE, 2020. doi:10.1109/SP40000.2020.00114.

23 Peter Schwabe, Benoît Viguier, Timmy Weerwag, and Freek Wiedijk. A Coq proof of the correctness of X25519 in TweetNaCl. In *Computer Security Foundations Symposium (CSF) 2021*, pages 1–16. IEEE, 2021. doi:10.1109/CSF51468.2021.00023.

24 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013. doi:10.1017/S0956796813000142.

25 Laurent Théry. Proving the group law for elliptic curves formally. Technical Report RT-0330, INRIA, 2007. URL: https://hal.inria.fr/inria-00129237.

26 Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Conference on Computer and Communications Security (CCS) 2017*, pages 1789–1806. ACM, 2017. doi:10.1145/3133956.3134043.