Foundational Integration Verification of a Cryptographic Server

ANDRES ERBSEN, Google, USA
JADE PHILIPOOM, Google, Germany
DUSTIN JAMNER, Massachusetts Institute of Technology, USA
ASHLEY LIN, Massachusetts Institute of Technology, USA
SAMUEL GRUETTER, Massachusetts Institute of Technology, USA
CLÉMENT PIT-CLAUDEL, EPFL, Switzerland
ADAM CHLIPALA, Massachusetts Institute of Technology, USA

We present verification of a bare-metal server built using diverse implementation techniques and languages against a whole-system input-output specification in terms of machine code, network packets, and mathematical specifications of elliptic-curve cryptography. We used very different formal-reasoning techniques throughout the stack, ranging from computer algebra, symbolic execution, and verification-condition generation to interactive verification of functional programs including compilers for C-like and functional languages. All these component specifications and domain-specific reasoning techniques are defined and justified against common foundations in the Coq proof assistant. Connecting these components is a minimalistic specification style based on functional programs and assertions over simple objects, omnisemantics for program execution, and basic separation logic for memory layout. This design enables us to bring the components together in a top-level correctness theorem that can be audited without understanding or trusting the internal interfaces and tools. Our case study is a simple cryptographic server for flipping of a bit of state through public-key authenticated network messages, and its proof shows total functional correctness including static bounds on memory usage. This paper also describes our experiences with the specific verification tools we build upon, along with detailed analysis of reasons behind the widely varying levels of productivity we experienced between combinations of tools and tasks.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Semantics**; *Integration frameworks*; • **Security and privacy → Public key (asymmetric) techniques**.

Additional Key Words and Phrases: proof assistants, elliptic-curve cryptography, bare-metal programming

ACM Reference Format:
Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Samuel Gruetter, Clément Pit-Claudel, and Adam Chlipala. 2024. Foundational Integration Verification of a Cryptographic Server. *Proc. ACM Program. Lang.* 8, PLDI, Article 216 (June 2024), 26 pages. https://doi.org/10.1145/3656446

1 INTRODUCTION

Formal verification can conclusively rule out deviations of a software component from its formal specification, and highly automated and effective verification tools are available for specific implementation and specification formats. However, specifications of computer-systems components

Authors' addresses: Andres Erbsen, Google, Cambridge, USA; Jade Philipoom, Google, Berlin, Germany; Dustin Jamner, Massachusetts Institute of Technology, Cambridge, USA; Ashley Lin, Massachusetts Institute of Technology, Cambridge, USA; Samuel Gruetter, Massachusetts Institute of Technology, Cambridge, USA; Clément Pit-Claudel, EPFL, Lausanne, Switzerland; Adam Chlipala, Massachusetts Institute of Technology, Cambridge, USA.

ACM 2475-1421/2024/6-ART216
https://doi.org/10.1145/3656446

(and even the languages they are written in) can be astoundingly subtle, leaving room for concern about implementation flaws simply being replicated in these specifications or the verification tools themselves. At the same time, the intuitive notion that the components of a system should "just work well together" is often much *easier* to formalize and yet hardly ever feasible to prove directly. We report on an experiment in *integration verification*, that is proving system correctness through composing modular component proofs, for a bare-metal software stack that includes functional and imperative languages, three qualitatively different compilers, optimized implementations of elliptic-curve and addition-rotate-XOR cryptography, an Ethernet driver, UDP/IP networking, and utility libraries along with a proof-of-concept application.

Our work is unique in achieving a machine-checked integrated correctness proof about *all software in a system* by building on completely different reasoning methods in different subdomains. A key challenge is creating formal specifications that delineate responsibilities between adjacent components and justifying the use of domain-specific reasoning as a step towards a precise theorem about input and output of the entire system. To enable rigorous yet flexible reasoning, we work in the Coq proof assistant, restricting ourselves to domain-specific tools that can generate proofs checkable by it. Conceptually, the program-verification, theory-solver, and computer-algebra tools compatible with Coq can be taken as stand-ins for their independent counterparts, and we also benefit from the flexibility of combining them in ad-hoc ways with incremental mechanized checks against corner-cutting. However, insisting on *foundational* verification (meaning that all component verifications must generate proofs in a common core language) also comes with challenges: the selection of proof-producing or proven-correct verification tools is limited due both to inherent technical challenges and skilled implementers prioritizing new research. (In particular, we make do without modern SAT or SMT solvers due to challenges in generating efficiently checkable proofs from them.) Thus we present separate conclusions about the conceptual plan, the specification style we converged on, and concrete use of the research artifacts we build upon.

To make the discussion more concrete, we will now describe the demonstration system we built and the theorem we proved about it. Yet we request that the reader keep in mind that this system exists exactly for concreteness – we built the demo to show off the integration of the components, not the components themselves. Our demonstration system is based on a SiFive FE310 microcontroller that runs RISC-V (RV32IM) [Waterman and Asanovic 2019] code in a bare-metal environment. The server listens for UDP packets over Ethernet, responds to a session initiation with an X25519 elliptic-curve Diffie-Hellman key exchange, and accepts a different packet type to complete the handshake and authenticate the user whose authorized public key is specified in the system configuration. After successful authentication, a general-purpose digital output is driven based on the command from the received packet. In the physical demonstrations, this output is connected to a motor controller that opens or closes a toy garage door, which stands in for remotely managed real-world infrastructure such as a power plant or the gates of a dam.

The specification is simple: successfully authenticated commands should drive the actuator, while other input should be ignored. The required elliptic-curve mathematics are proven against affine-coordinate formulas that fit on the back of a napkin, and the network packet formats are specified by concatenating appropriate lists of bytes in a functional language. The bottom interface is RISC-V machine code. Someone who wishes to understand and audit the *statement* of our system theorem does *not* need to read definitions of the programming languages, APIs and ABIs, and resource accounting we use that ensures that the system will not run out of memory, overwrite code with data, or enter an infinite loop, because our *statement* does not include or reference any definitions related to these concerns. The *proof* of our statement has to deal with all these concerns, but the internal specifications all "cancel out" as we compose the subproofs.

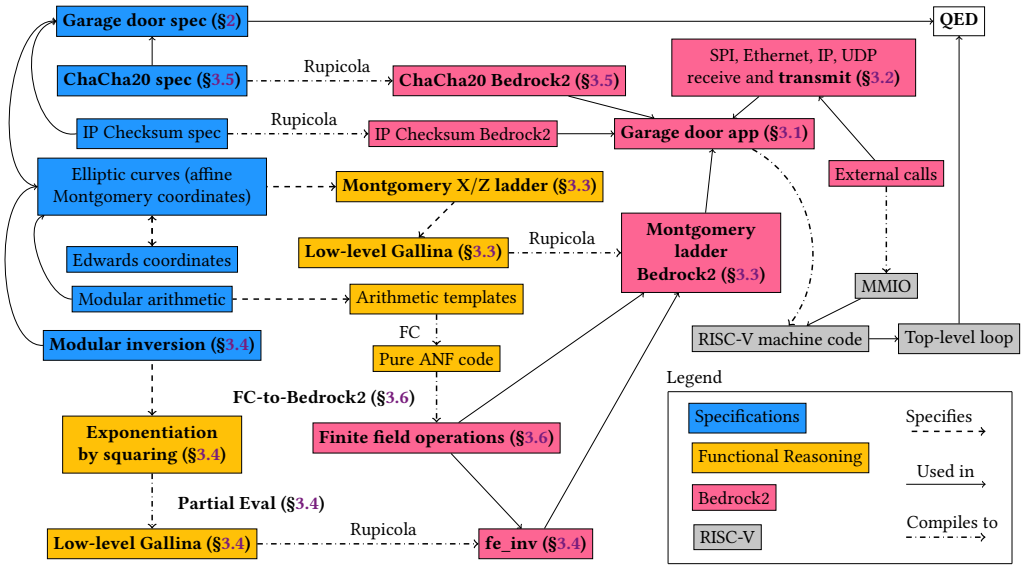

Fig. 1. System diagram. Contributions of this work are bolded.

Our implementation builds upon established Coq projects from each domain, reusing both methods and artifacts when practical while extending each project as required for verified integration:

- Underspecification and unconditional requirements in internal specifications are encoded using omnisemantics [Charguéraud et al. 2023], and we use simple separation logic as an assertion language throughout the stack, even to specify compilers.
- We build upon the Bedrock2 [Erbsen et al. 2021] framework. Its C-like language and proof and compilation tools are the core of our approach, and we show that it can be applied to a significantly more complex specification, with network input and output, compared to past work. We also add support for reasoning about read-before-write aliasing.
- We adopt the Fiat Cryptography [Erbsen et al. 2019] framework, for generation of fast finite-field arithmetic from templates expressed and proven as higher-order functional programs. We extend it *above* with proofs of elliptic-curve-point representations structures and algorithms, as well as *below* with a new verified backend targeting Bedrock2.
- We adopt the Rupicola [Pit-Claudel et al. 2022] code generator, deriving bare-metal-ready Bedrock2 code from functional programs. We use it to build cryptographic code on top of finite-field arithmetic and also integrate code for IP checksums from past work on Rupicola.
- We experimented with an interactive-proof-driven variant of relational compilation for generating code for algorithmically straightforward routines with memory-access-related reasoning bottlenecks such as ChaCha20.

Most importantly, we pull all components (Figure 1) together in a concrete case study of a bare-metal networked cryptographic system, with full integration of these tools and their proofs into a machine-checked end-to-end theorem (Figure 2) that connects the behavior of a piece of RISC-V machine code to a high-level mathematical specification of cryptographic algorithms. We achieve a trusted code base (TCB) significantly smaller than in past work for systems of this complexity level, while allowing for use of high-level languages for challenging implementation tasks. Section 2 explains our main theorem and integration-verification story, leaving Section 3 to present our implementation's components in more detail. Section 4 evaluates our artifact and approach. Section 5 reviews the state of the art in comprehensive verification, and Section 6 concludes.

```
Definition initial_conditions mach := 0x20400000 = mach.(getPc) ∧ [] = mach.(getLog) …
Theorem garagedoor_correct : forall mach : RiscvMachine, initial_conditions mach ->
  always run1 (eventually run1 (fun mach' => io_spec mach'.(getLog))) mach.
```

Fig. 2. End-to-end theorem relating I/O behavior of RISC-V machines starting from states satisfying `initial_conditions` to the network protocol described by the trace predicate `io_spec`.

Our implementation and proofs are available under a permissive open-source license.[1]

2 INTEGRATION VERIFICATION FROM MACHINE CODE TO NETWORK PACKETS

Figure 2 shows the statement of the end-to-end theorem that we proved about our demonstration system. This section will unpack the definitions appearing in that statement and review the basic ingredients used to specify the functional correctness of components throughout the system.

The foundation of our verified stack is a model of the RISC-V instruction set. Specifically, `run1` represents the execution of a single RISC-V instruction as a predicate transformer: it takes a postcondition as an argument and returns the proposition that must hold one instruction earlier. Runtime input is modeled by `run1` invoking its postcondition under a universal quantifier. The definition of `run1` instantiates an executable RISC-V model with omnisemantics for input nondeterminism [§3.4.5 Bourgeat et al. 2023; Erbsen et al. 2021]. So we can state properties of interaction of the RISC-V processor with the outside world, our model of the processor includes the trace of all inputs and outputs observed during its execution as a piece of ghost state.

Our system theorem covers the execution from the first programmable RISC-V instruction onwards, but it relies on specific conditions on the initial state: the program counter must start out pointing to the address where the verified machine code begins, the ghost-state trace must start out empty, the required amount of memory must be available, and so on. In return, our specification guarantees liveness and crash-freedom: `io_spec` will be repeatedly reestablished without any carve-outs for potentially running out of memory or entering a silent infinite loop instead.

This liveness guarantee is encoded using the combinators `always` and `eventually` inspired by the temporal-logic operators □ and ◊: `always run1 P mach` means that `P` holds on the machine state `mach` and that after `run1`, `always run1 P` holds in any state that might result. Similarly, `eventually run1 P mach` means that `P` holds on `mach` or `eventually run1 P` holds after `run1`, with input received during `run1` potentially influencing the number of steps until reaching `P`.[2]

While `always` in the specification simply encodes that we are proving an invariant, `eventually` allows us to use a specification that only applies after having received an entire network packet and potentially responded to it, but not in the middle of the transaction. As our system's response to one packet may depend on past inputs, we extend the trace-predicate formalism from Erbsen et al. [2021, §3.1] with stateful protocols encoded as labeled transition systems. For the demo:

```
Record state := { prng_state : list byte ; x25519_ephemeral_secret : list byte }.
Definition protocol_step : state -> list MMIO -> state -> Prop := (* state machine *).
Definition protocol_spec l := exists s s', labeled_transitions protocol_step s s' l.
Definition io_spec: list _ -> Prop := only_mmio_satisfying (boot_seq +++ protocol_spec).
```

Note that `state` is for protocol specification only; `io_spec` does not (and cannot) say how it is represented in machine memory. The state-machine style is only used for the network protocol; peripheral initialization preceding it (`+++`) is still specified using simple regex-like predicates. Both

[1] https://github.com/mit-plv/foundational-integration-verification-of-a-cryptographic-server

[2] For `always run1 P`, we give and prove equivalent a coinductive definition and a characterization through the existence of an inductive invariant that implies `P` [Erbsen 2022, §4.4.3]. `eventually` is defined inductively [§3.2 Charguéraud et al. 2023; Erbsen et al. 2021, §4.3]. We also proved that the trace itself can always be extended to satisfy `io_spec`.

phases use memory-mapped input and output (MMIO), so each trace element is a read or a write with an address and a value (other work in the same framework also uses direct memory access).

2.1 Network Protocol Specification

The possible inputs, outputs, and transitions of our simple network protocol are specified in 50 lines of ad-hoc casework making up the relation `protocol_step`. Protocols featuring complex state diagrams, or sequential or parallel composition of subprotocols, would likely benefit from expansion of the verified stack up towards simpler top-level specifications, perhaps by integrating a certifying compiler as in Ikebuchi et al. [2022], or verification of distributed-systems properties.

However, regardless of how protocol-level control flow is expressed, the specification must relate inputs to computed values and output actions, which we will sketch in this section by showing a cross-cutting sample of definitions that specify the server sending its first packet in the protocol.

In our demo, the output can either be a network packet or an actuation of a general-purpose digital output pin controlling a motor that raises or lowers the door (Figure 3). Both are encoded as MMIO writes to appropriate peripherals. For example, transmitting some bytes over Ethernet is specified as a sequence of MMIO writes, using `le_combine` to respect the little-endian interface:

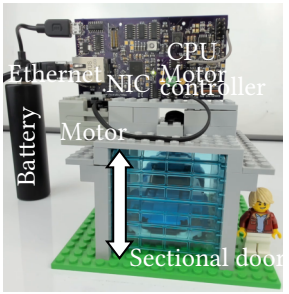

Fig. 3. Physical demonstration

```
Fixpoint lan9250_writepacket (bs : list byte) : list MMIO -> Prop := match bs with
  | [] => fun trace => trace = []
  | b0::b1::b2::b3::bs => lan9250_writeword TX_DATA_FIFO (le_combine [b0;b1;b2;b3])
                          +++ lan9250_writepacket bs
  | _ => fun _ => False end.
```

The contents of the network packets are described using simple functional-program expressions inside `protocol_step`. For example, here is our specification of the server's first packet of a Diffie-Hellman handshake featuring headers of network protocols and a fresh X25519 public key.

```
Ethernet header { mac_remote ++ mac_local ++ be2 ethertype ++
IP header       { let ip_hdr checksum := ih_const ++ be2 ip_length ++
                                         ip_idff ++ [ipproto] ++ le_split 2 checksum ++
                                         ip_local ++ ip_remote in
                ip_hdr (IPChecksum.Spec.ip_checksum (ip_hdr 0)) ++
UDP header      { udp_local ++ udp_remote ++ be2 udp_length ++ be2 0 ++
Application data { garagedoor_header ++
                    x25519_spec x25519_ephemeral_secret Curve25519.M.B
```

The elliptic-curve payload of the packet is described using the same building blocks for encoding and decoding as well as domain-standard mathematical constructions such as getting the `X` coordinate of a point, unary integer-point multiplication, and forcing a number to be 0 mod 8:

```
Definition x25519_spec s P :=
  le_split 32 (M.X0 (Curve25519.M.scalarmult (Curve25519.clamp (le_combine s)) P)).
```

The point-addition function from whose repetition multiplication arises is specified using standard affine formulas[3] (shown here in twisted Edwards coordinates for brevity and proven equivalent to the Montgomery version):

[3] https://en.wikipedia.org/wiki/Twisted_Edwards_curve#Addition_on_twisted_Edwards_curves

```
Program Definition add (P1 P2 : point) : point :=
  match coordinates P1, coordinates P2 return F*F with (x1, y1), (x2, y2) =>
    (((x1*y2+y1*x2)/(1+d*x1*x2*y1*y2)), ((y1*y2-a*x1*x2)/(1-d*x1*x2*y1*y2))) end.
```

These specifications are illustrative of a more general pattern that also persists throughout the internal interfaces of our system: we state requirements directly in terms of well-known operations on simple objects. In the packet-content specification we see integers ($\mathbb{Z}$) translated to little-endian (`le_split`) or big-endian (`be2`) lists of bytes that are then concatenated (`++`). An ad-hoc pure function describes the content of the IP header before and after checksum computation. In `add`, `F` refers to a field that is later instantiated with $\mathbb{Z}/(2^{255}-19)\mathbb{Z}$ and serialized using `le_split`. We are highlighting this barebones style in contrast to defining a more specialized framework for packet formats or cryptographic computations: simple functional programs work well enough and are suitable for use as specifications of the packet-processing code written in Bedrock2, Rupicola's IP-checksum example, and elliptic-curve arithmetic from Fiat Cryptography.

2.2 Common Concepts Throughout Component Specifications

Specification technique: we adopt and recommend stating specifications using functional reference implementations for computation, simple separation logic for memory layout, and omnisemantics for execution. Traces and trace predicates to specify input and output also worked well given our goal to prove continued correct operation regardless of invalid input.

The pursuit to verify systems of several components naturally tempts the forward-looking architect to consider intentionally introducing uniformity to how the parts are implemented, specified, and verified. Indeed, as each gap and idiosyncratic choice that does not appear in the top-level theorem needs to be reconciled by proof, uniformity can straightforwardly reduce accidental complexity and associated toil. Further, there is a common intuition behind many successful programming-languages techniques that introducing structure (constraints) can then simplify working with programs expressed within these constraints. For systems verification specifically, a number of pervasive frameworks have been proposed: certified abstraction layers enforce contextual equivalence of implementations [Gu et al. 2016], interaction trees can be used to reason equationally about nested reactive behaviors [Koh et al. 2019], Iris can be used to verify safety of particular concurrent combinations of programs [Jung et al. 2015], and so on. Substantial case studies within each framework have shown impressive integration successes, and yet none of the past whole-system proofs surveyed in section 5 use these highly structured ecosystems, instead relying on purpose-built components.

We integrate three independently successful projects that were not developed within a single prescriptive framework, and yet we find that their most primitive specifications (once all domain-specific machinery has been encapsulated) are enough to connect them with manageable glue code and integration proofs. For example, while Fiat Cryptography relies on parametric templates and a verified higher-order partial-evaluation engine under the hood, the generated straight-line three-operand code and specifications like $\lfloor$`mulmod a b`$\rfloor = \lfloor a \rfloor \cdot \lfloor b \rfloor \bmod m$ are not tied to these tools. Similarly, Rupicola proves that code generated through relational compilation implements the purely functional input program. Rather than *adding* structure throughout, we *reduce* project-specific definitions to their basic Coq building blocks and rely on that common foundation for integration at point-in-time interfaces such as function boundaries.

Three specification considerations appear throughout the components of our stack: modeling execution of imperative languages with undefined (forbidden) and unspecified (nondeterministic) cases, passing data by reference to addressable memory of these languages, and keeping track of interaction with the outside world. We made consistent but unambitious choices, intentionally favoring flexibility in case of unforeseen challenges over general properties from curated structure:

(1) We specify programming languages that do not have straightforward denotations to Coq using operational semantics. Per our goal to prove total-correctness properties of all possible executions, we use the omnisemantics flavor [Charguéraud et al. 2023], which amounts to weakest-precondition predicate transformers defined inductively.
(2) Preconditions and postconditions of impure programs relate functional reference models of their arguments and return values to in-memory representations thereof using separation logic, regardless of whether these programs are written by hand or generated by a compiler. Most specifications use separating conjunction `*` as a simple disjoint-union operator; in some cases we also use existential quantifiers for data abstraction and non-separating conjunction for read-before-write aliasing.
(3) I/O is modeled using lists of language-appropriate events as in Erbsen et al. [2021].

While the structure of our specifications is uninventive, complex reasoning can be required to satisfy them. The specifications of efficient algorithms for computing with mathematical objects are described in terms of naive purely functional descriptions of the same operations. Thus the specifications are computable in the theoretical sense, but actually running them in practice is often infeasible: for example, sometimes multi-hundred-bit natural numbers are treated as unary. Similarly, reconciling shallow differences between different libraries' specifications of the same concept is accomplished as a part of proving the integration between them, sometimes restating the specification of one using the definitions from the other. Doing so is a chore, but it works.

2.3 Read-Before-Write Aliasing in Separation Logic

We make important use of non-separating conjunction in a few places to state preconditions of functions whose inputs may overlap each other or the output. C's `memmove` is a canonical example:

```
fnspec! "memmove" (dst src n : word) / (d s : list byte) (R Rs : mem -> Prop),
{ requires t m := m =* bytes src s * Rs ∧ m =* bytes dst d * R ∧
                  length s = n :> Z ∧ length d = n :> Z ∧ n <= 2^31;
  ensures t' m := m =* bytes dst s * R ∧ t=t' }.
```

The function takes two pointers and a length as arguments, assumes that each pointer points to a list of bytes (ghost arguments `d` and `s`), and overwrites the destination with the source, taking care to catch the entire input even if it overlaps the output. Critically, the specification here uses separate frame quantifiers (`R` and `Rs`) in the two non-separating conjuncts asserted on the same memory (`m =*`), and only the frame `R` of the destination buffer is preserved in the postcondition. Depending on how the function is called, `R` may contain (part of) the source buffer again, in which case that part is preserved, or not. The source frame `Rs` is intentionally unused in the postcondition to indicate that it may not be preserved. We usually abbreviate such preconditions as `m =*> bytes src s`, which stands for `∃ Rs, bytes src s * Rs` to avoid naming that frame.

Both proof automation for handwritten imperative code and compilers generating imperative code support calling[4] functions specified in this manner at the same level of automation as other separation-logic preconditions: each non-separating conjunct is solved independently, instantiating each frame quantifier through cancellation. For instance, consider a (no-op) call to `memmove` with `dst = src`, so cancellation finds `d = s` and `Rs = R`. In this case, the two non-separating conjuncts in the precondition are identical, and the specification overall describes an identity function. Yet Fiat-Crypto-generated three-operand arithmetic routines whose output is some nontrivial function of the input are useful even in the case where both inputs match the output: for example, an addition function can be used to perform in-place doubling. On the other hand, calling `memmove`

[4]The proof of `memmove` itself is less streamlined and relies on Bedrock2 (unlike VST and CompCert) giving the simple, permissive semantics to compare addresses of memory potentially returned by different allocations.

with disjoint inputs results in each input being inferred to belong to the frame of the other, which again makes the two non-separating conjunctions in the precondition identical but represents the same guarantee as would be derived from a specification that used a single frame and two buffers in both the precondition and the postcondition.

This simple specification style proved convenient for the variety of components that we verified. As usual in Coq, it is important that we can mention native Gallina functional programs of arbitrary sophistication, anywhere within separation-logic specifications, "for free." Those programs wind up translated to Bedrock2 code in a variety of ways, all of which turned out to be compatible with this style. We carried out such proofs directly on manually written Bedrock2 programs, integrated with four different verified or proof-generating translators (two new to our project), and connected this specification style in the end with the Bedrock2 compiler's proof.

We now turn to details of those different components.

3 INTEGRATING COMPONENT PROOFS

3.1 RISC-V Machine Code for Memory-Mapped I/O and Infinite Loops

The lowest-level language considered in this study is machine code for the RISC-V instruction set. We follow the specification and reasoning strategy from Bedrock2 [Bourgeat et al. 2023; Erbsen et al. 2021] but target the RV32IM instruction set of a commercial microcontroller instead of a verified processor implemented on an FPGA. While mainstream C code commonly uses inline assembly and macros to access hardware features not supported by the compiler, the ad-hoc RISC-V machine code used for similar purposes in the work we build upon is held to the same standards of verification as the compiler-generated code.

Specification technique: treat inline assembly blocks as mini-compilers to generate verified assembly for those parts not directly expressible in the Bedrock2 source language.

The Bedrock2 compiler is proven to satisfy omnisemantics specification preservation. The statement that a program c satisfies a postcondition Q is written $c \Downarrow Q$ and captures safety, termination, and an overapproximation of the set of possible outcomes. Then an optimization pass f should satisfy $c \Downarrow Q \rightarrow f(c) \Downarrow Q$, and a compiler between languages with different state representations needs a conclusion with a Q' defined in terms of Q. In this context, Q' includes invariants about RISC-V registers, a decomposition of the machine memory as a disjoint union of source-language memory and compiler-managed data structures such as function activation records on the stack, and a mapping between Bedrock2 external-call traces and RISC-V memory-mapped I/O (MMIO) traces. Compiler-correctness proofs proceed by induction on $\Downarrow$ under suitably strengthened hypotheses about the initial state [Erbsen et al. 2021, §4.2] [Charguéraud et al. 2023, §6].

The first problem solved using a mini-compiler is that our use of memory-mapped I/O to talk to devices is not directly expressible in the Bedrock2 source language. Instead, it is modeled by external calls, and the Bedrock2 compiler is parameterized over a mini-compiler for these external calls, which replaces each call by either a load or store assembly instruction. Thus, the proof of the assembly fragment for an external call must centrally establish the relationship between the low-level and high-level I/O-trace events, but it must also show that the MMIO store does not accidentally overwrite compiler data structures.

Moreover, the top-level loop is an infinite loop, but the Bedrock2 source-language semantics only accept terminating programs, so we use another verified assembly macro that takes the relative addresses of an init function and a loop-body function, emitting a program that first calls the init function and then repeatedly calls the loop-body function forever. In this case, the specification of the assembly fragment is parametrized over that of the loop body, and this specification states that any loop invariant is always eventually satisfied, leading to the top-level spec in Figure 2.

We reuse both mini-compilers from Erbsen et al. [2021]. Their specifications are considerably longer than their implementations, and the corresponding proofs require correspondingly detailed ad-hoc reasoning, but there are no surprise obstacles or hard-to-bridge abstraction gaps. These proofs are made feasible by the use of language-independent separation-logic assertions and straightforward elementwise relations connecting Bedrock2 and RISC-V-level I/O traces. These techniques enable us to compose an end-to-end theorem that covers ad-hoc assembly code for functionality without language support, as opposed to relying on some unverified assembly glue code or extending the syntax and semantics of the languages.

3.2 Driver and Packet-Processing Verification in Bedrock2

Specification technique: next lowest-level functions written in a C-like language are specified using omnisemantics with separation-logic and trace-predicate assertions.

Again following past work, the Bedrock2 language (a simple well-defined subset of C) is at the center of our system. However, most code in our system is not written in Bedrock2 either: we only use it directly for I/O-heavy and computationally straightforward routines such as drivers and packet shuffling, preferring functional programming otherwise. Nevertheless, all higher-level compilers use Bedrock2 as a target language, so the vast majority of code in our system is untimately compiled by the Bedrock2 compiler, appealing to the compiler's correctness proof.

To understand how computation-centric specifications interface with the I/O specifications, consider the specification of the Ethernet transmit routine in our driver:

```
fnspec! "lan9250_tx" p l / bs ~> err,
{ requires t m := m =*> bytes p bs ∧ l = length bs :> Z ∧ l mod 4 = 0 :> Z;
  ensures T M := M = m ∧ ∃ t', T = t' ++ t ∧ only_mmio_satisfying (fun h =>
    (0 <> err ∧ (any +++ spi_timeout) h) ∨ (0 = err ∧ lan9250_send bs h)) t' }.
```

This specification style will appear uniformly throughout our case study, so it is worth explaining a few of the most important structural elements. Here we are indicating that function `lan9250_tx` receives two parameters named `p` and `l`, in addition to one *ghost parameter* `bs`, which is not passed at runtime but included to specify the relationship between the memory input and I/O trace. The ghost parameter is separated from the other parameters with `/`. Then we have a precondition in a `requires` clause and postcondition in an `ensures` clause. The precondition may mention *initial* I/O trace `t` and memory `m`, while the postcondition may mention those variables plus their *final* counterparts `T` and `M`. In terms of the omnisemantics specification of Bedrock2, the `ensures` clause is the outcome set in which all executions must terminate, program and ghost variables bound after `fnspec!` are universally quantified, and `requires` represents hypotheses assumed about them – informally, ∀ args. `requires` → f(`args`) ⇓ `ensures`.

In a scenario where the packet to be transmitted contains a Diffie-Hellman public key `pk`, the specification would be instantiated with `bs := pk`. Thus, the precondition `requires` that the memory `m` contains the bytes `pk` of length `l` at the address `p`. The postcondition promises that the entire memory remains unchanged (regardless of whether transmission succeeded) and that the return value `err` corresponds to the I/O trace of the function: either a timeout was reached, or the exact bytes `bs = pk` were transmitted.

The implementation loops over the 4-byte words in the packet, relying on the `l mod 4` precondition to ensure that all bytes in the packet buffer `p` have been transmitted once the loop exits:

```
Definition lan9250_tx := func! (p, l) ~> err {
  err = lan9250_writeword($TX_DATA_FIFO, $(2^13)|$(2^12)|l); require !err;
  err = lan9250_writeword($TX_DATA_FIFO, l);                 require !err;
  while ($3 < l) {
```

```
unpack! err = lan9250_writeword($TX_DATA_FIFO, load4(p));
if err { l = $0 } else { p = p + $4; l = l - $4 } } }.
```

Building blocks such as `lan9259_writeword` are themselves implemented similarly, bottoming out in mini-compiler operations like `MMIOWRITE(0x10024048, b)` whose semantics extend the I/O trace `T` with the arguments and return values. Trace predicates such as `lan9250_send` used in the specification mirror the control flow of the implementation, but they lack opportunities for undefined behavior, are oblivious to memory layout, and use integers instead of machine words. (Specifically, `lan9250_send` describes how the packet length and metadata are communciated and then defers to `lan9250_writepacket` from Section 2.1.)

Discussion. The Bedrock2 toolchain comes with a program logic based on omnisemantics and separation-logic assertions, featuring a selection of loop-invariant rules, symbolic execution for straight-line code with only simple memory accesses, glue tactics for using Coq's linear-arithmetic prover with machine-word arithmetic from Bedrock2, and tactics for matching up I/O traces found in program specifications and symbolic-execution output. As Bedrock2 is intended to support verification of very tricky programs, the emphasis is not on full automation but rather on taking care of the boring parts of program proof while predictably generating tractable Coq goals (with the proof context representing the symbolic state) for cases that cannot be solved automatically.

We found the proof tooling at this level to be okay. It is fairly common to find that simple-looking side conditions are not proved automatically, but one can generally push predictably through these proofs with tactics specialized to the Hoare-logic style. The main units of work and our time spent on them are specification and proof of `memmove` (2 days), proof of `memequal`/`memswap`/`memconst` (1 day), translating from C and proving an improved Ethernet driver (2 days), and generalization of the compiler integration proofs from Erbsen et al. [2021] (2 weeks).

After the pieces discussed in the rest of the paper were in place, one team member spent two weeks at 50% effort implementing the network protocol and integrating the components, so that the full physical demo ran; then another two weeks at 50% time were spent integrating the proofs into our top-level theorem. We also worked on X25519-specific proof integration over two to three weeks. We do want to emphasize that there was substantial heterogeneity in who was proving different components (not just in the compilation/proof styles used therein), so this integration effort feels fairly realistic for a basic-research study.

3.3 Allocation-Free Functional Programming with Rupicola

Specification technique: functional programs are natural specifications for most imperative computations, and C-like implementations with simple memory access but nontrivial control flow can be derived from functional programs automatically.

We use Rupicola to translate functional code into imperative code. Rupicola implements the concept of relational compilation [Pit-Claudel et al. 2022], the idea that given a source program s, correct-by-construction compilation can be seen as a kind of proof search for proofs of $\exists t.\ R(s,t)$, where R is a relation between source and target programs that serves as the specification of t. Rupicola performs this proof search via Ltac automation using an extensible corpus of compilation lemmas. Each lemma describes the relationship between one component of the source program, say an addition or a `map` function invocation, and a snippet of imperative code that implements it. The relation given by the pool of lemmas is generally partial because not all Gallina operations will match compilation lemmas from the library, and the relation can be many-to-many when multiple lemmas match the same source. For example, there might be one lemma that compiles a `map` to a while loop, while another unrolls `map`s over known-length lists. The Rupicola compilation process is designed for high automation, but its success in this regard can vary based on the automation's

ability to solve side conditions and the difference between the structures of the compilation lemmas and that of the program under consideration.

Mechanically, Rupicola takes advantage of Coq's `Derive` command, which allows developers to simultaneously construct a definition and a proof of a property of that definition using tactics. A standard Rupicola derivation defines a block of Bedrock2 code this way, where the predicate it verifies is a Hoare triple in the Bedrock2 program logic where the postcondition describes the expected output state of the program in terms of a related Gallina function, and the precondition characterizes the valid inputs to that function. Rupicola then instantiates the Bedrock2 code block by applying lemmas that fit the structure of the Gallina function and unifying their associated Bedrock2 snippets with subterms of the code being derived.

This style is preferred for new code in our system, though it is not universally applicable. In the more straightforward cases, we feed a functional specification directly to Rupicola and can derive performant imperative code. If substantial concessions are required to express the data flow in a manner that makes translation to imperative code straightforward, we prove a low-level functional implementation against a higher-level spec and then derive the imperative code from the low-level version. This higher-level spec could either be a readability-oriented version of the low-level code or an independent description of what the algorithm is supposed to accomplish. For example, our Montgomery ladder is proven to compute the Montgomery x coordinate of elliptic-curve scalar multiplication, specified in affine coordinates. We also reuse some existing Rupicola code and proofs verbatim to compute IP checksums.

Functional Programming and Computer Algebra. Implementations of elliptic-curve operations contain sophisticated algorithms with large algebraic expressions and subtle case distinctions. Efficiency considerations require the use of redundant representations: for a straightforward example, a two-coordinate elliptic-curve point may be represented using three elements of the underlying field. We do not want to tackle the proof challenges in tandem with tracking pointers through imperative code. Instead, we prefer to verify functional programs and then compile them effectively.

```
Definition montladder (scalarbits : Z) (testbit:Z->bool) (x1:F) : F :=
    (while (fun '(_, i) => Z.geb i 0)
    (fun '(x2, z2, x3, z3, swap, i) =>
      dlet b := testbit i in dlet swap := xor swap b in
      let (x2, x3) := cswap swap x2 x3 in let (z2, z3) := cswap swap z2 z3 in
      dlet swap := b in let '((x2, z2), (x3, z3)) := xzladderstep x1 (x2, z2) (x3, z3) in
      let i := Z.pred i in (x2, z2, x3, z3, swap, i))
    (Z.to_nat scalarbits) (1%F, 0%F, x1, 1%F, false, Z.pred scalarbits))
```

Fig. 4. Montgomery-ladder body in Gallina

Figure 4 shows an excerpt from the Montgomery-ladder algorithm for computing the scalar-point multiplication in the case where neither input is known at compile time and must not be leaked through timing side channels. The proof that this function corresponds to a specification in terms of unary multiplication ($(n+1)P = nP+P$) and standard point-addition formulas in affine coordinates is straightforward using a combination of case analysis and Gröbner-basis computations, taking a Coq novice only a few days. In contrast, separation-logic-based verification of sophisticated algorithms encoded in an imperative language is believed to take at least a semester's worth of training and still sometimes avoided. For example, to verify the TweetNaCl implementation of Curve25519 using VST [Schwabe et al. 2021], a Coq program that closely mirrored the C code was first written and proven against it, to be then analyzed further with imperative-language concerns

```
fnspec! "montladder" pOUT pK pU / Kbytes (K : Z) (U : F _) OUT R,
{ requires t m := m =* FElem pOUT OUT * bytes pK Kbytes * FElem pU U * R ∧
    le_combine Kbytes = K ∧ scalarbits <= 8*(length Kbytes) :> Z;
  ensures  T M := T = t ∧ (let OUT := montladder_lowlevel M_pos a24 scalarbits K U in
                 M =* FElem pOUT OUT * bytes pK Kbytes * FElem pU U * R) }.

Derive montladder_body SuchThat
  (defn! "montladder" ("OUT", "K", "U") { montladder_body },
    implements montladder_lowlevel using [felem_cswap; felem_copy; from_word; (*...*) ])
  As montladder_correct. pose proof scalarbits_bound.  compile.  Qed.
```

Fig. 5. Montgomery-ladder implementation correctness

```
stackalloc 40 as X2; fe25519_copy(X2, U);
stackalloc 40 as Z2; fe25519_from_word(Z2, 1);
swap = 0; i = 255;
while 0 < i {
  i = i - 1; s_i = load1(K + i>>3)>>(i & 7) & 1;
  swap = swap ^ s_i; felem_cswap(swap, X1, X2); felem_cswap(swap, Z1, Z2);
  ladderstep(U, X1, Z1, X2, Z2); swap = s_i; unset s_i }
```

Fig. 6. Montgomery-ladder imperative code

out-of-the-way. In our case, the functional-language description of the optimized algorithm serves as the input to derivation of Bedrock2 C code using Rupicola.

As the same functional program could be matched by multiple imperative programs, we write a low-level version of it that includes hints about desired imperative-language implementation choices. What distinguishes that low-level functional code from the high-level code of Figure 4 is that the low-level version uses special identity functions to annotate the choice between in-place operations and stack allocations of temporaries. Algorithmically, the two are identical, and we prove them equivalent using standard Coq proof techniques, since they are both written as functional programs in Coq's built-in language.

In Figure 5, we show the theorem and proof script that derives the imperative code excerpted in Figure 6 and simultaneously proves its correctness with respect to the low-level functional code. We first add a necessary assumption to the context (`scalarbits` $< 2^{32}$) and then use Rupicola's `compile` tactic to handle the proof. The `compile` tactic will then apply the compilation lemma associated with each function call in the order that they occur in the program, solving side conditions with various ancillary tactics as they arise. Specifically, here Coq's linear-arithmetic solver proves `i>>3` stays within the bounds of the array starting at `pK`.

3.4 Template Metaprogramming With Rupicola

One of the key primitives missing from previous Fiat Cryptography work in order to derive a Montgomery ladder is modular inversion (i.e., inversion in the field $\mathbb{Z}/(2^{255}-19)\mathbb{Z}$). For this part, we use a combination of metaprogramming, partial evaluation, and relational compilation to generate a Bedrock2 implementation using addition-chain exponentiation. For example, for $x^{-1} \bmod 89$:

(1) We start with a Coq spec of modular inversion. $x \mapsto x^{-1} \pmod{89}$
(2) Using Euler's theorem, we rewrite modular inverse into (naive) modular exponentiation. $x \mapsto x^{87} \pmod{89}$

(3) We then replace naive exponentiation with a call to a generic exponentiation-by-squaring algorithm (which approximates the best addition chain), implemented as a recursive Coq function. The code is still purely functional, but its structure is close to the one we want for the final program: internally, at run time, it computes a run-length encoding of the exponent's base-2 representation and uses a higher-order iterator `repeat` to perform repeated squares (for sequences of zeroes) or squares-and-multiplies (for sequences of ones).

$$x \mapsto \texttt{exp\_by\_sq\_mod}\ x\ \texttt{87}\ \texttt{89}$$

(4) Using Coq's partial-evaluation capabilities, we precompute the base-2 decomposition (for example, $87 = \texttt{0b1010111}$) and unroll the recursion: $x \mapsto (((((x^2)^2 \cdot x)^2)^2 \cdot x)^2 \cdot x)^2 \cdot x$ ... except that we in fact keep the `repeat` iterator and intermediate let-bindings, to get a purely functional program that implements exponentiation as a flat sequence of let-bindings (each corresponds to one or multiple square or square-and-multiply operations).

$$x \mapsto \texttt{(let } r \texttt{ := } x^2 \texttt{ in let } r \texttt{ := } r^2 \cdot x \texttt{ in let } r \texttt{ := } r^2 \texttt{ in let } r \texttt{ := repeat (} r \texttt{ => } r^2 \cdot x \texttt{ \% 89)} \ r\ \texttt{3 in}\ r\texttt{)}$$

(5) Finally, we use Rupicola to translate this pure program to Bedrock2. We map the higher-order iterator to a loop and squares and multiplies to optimized in-place implementations.

```
void inv89(ptr x, ptr r) {
  square89(r, x); square89(r, r); mult89(r, x, r); square89(r, r);
  for (int i = 0; i < 3; i++) { square89(r, r); mult89(r, x, r);} }
```

With this approach we do not have to write the Bedrock2 code by hand, yet we retain complete control over its generation. For example, using `repeat` in the implementation of step 3 gives us a more complex but smaller program. This approach squares nicely with standard Coq techniques, so that a new example should take an expert just a few hours of work. However, we assigned this part to a new team member as a learning-Coq exercise, and it took that person significantly longer.

3.5 Interactive Relational Compilation With Challenging Memory Accesses

After successfully using Rupicola to compile the previous examples, we went on to apply it to functions that operate on memory buffers, specifically ChaCha20. While we completed the verification, we did not achieve the same success as with X25519 due to two facets of the interaction between ChaCha20's specification and the Rupicola implementation's current design decisions. First, substantial quantities of side conditions escaped solution by the corpus of tactics that Rupicola invokes. For example, shifting between by-word and by-byte views of data required significant manual effort. Verifying such side conditions can impose a heavy burden when they do not fit within the capabilities of the automation. This problem is hardly unique to relational compilation, but it especially stands out when using a framework designed for high levels of automation like Rupicola. In particular, the pattern of applying a series of compilation lemmas tends to generate a large number of similar subgoals, so when they must be solved manually, it becomes a burden.

Additionally, an initial reliance on existing Rupicola mechanisms hindered our ability to express the low-level code in the best way and thus delayed development. Specifically, since Rupicola lemmas are structured following the design of Bedrock2, they handle temporary variables separately from memory. This dichotomy meant that existing Rupicola lemmas operate only on data in memory or variables without the ability to interchange them, even though in the high-level language it is natural to express operations on both with the same constructs.

We began ChaCha20 verification using the tools we had but found that performing certain operations in memory rather than in temporary variables necessitated extra stack allocation, which added complexity to the verification effort, including even more side conditions, and resulted in subpar generated code. Instead of finishing this approach, we reduced our dependence on existing

Rupicola idioms and implemented a separate set of lemmas, applying the idea of relational compilation to the domain of (fixed-length) arrays. Just as with Rupicola's expression and statement compilation lemmas, our array-programming extension to the relational-compilation pipeline involves building a set of verified local-variable array operations. We prove that new operations satisfy the `locals_array_expr` predicate, which takes as arguments preconditions on the program state and three equal-length lists, one of variable names, one of Bedrock2 expressions, and one of Gallina computations. This predicate connects to existing Rupicola idioms using a standard statement-level Rupicola compilation lemma [Pit-Claudel et al. 2022] that generates a `locals_array_expr` subgoal, shifting the derivation into the array-operation modality. When we apply this lemma in a derivation, it inserts the Bedrock2 code generated from the array operations by folding over the list of variable names and the list of Bedrock2 expressions associated with our high-level array operations and produces an unrolled sequence of assignment statements.

While our second approach proved successful, it still exposed some deficiencies in our current relational-compilation setup. Rupicola's design takes the opinionated stance that, with the right compilation lemmas in scope, program derivation should be (mostly) automatic. In theory, it degrades gracefully, in that the Rupicola compilation pipeline will proceed until it reaches a goal that it cannot solve and then offer that goal to the user. In practice, however, programs that trigger this behavior tend to require extensive supplemental proof effort.

In summary, Rupicola performed well on problems that shared a certain degree of similarity with the examples initially used to guide its design [Pit-Claudel et al. 2022, 2020]. However, Rupicola's current implementation did not suit the derivation of ChaCha20 well, especially with regard to integrating with a more manual approach to relational compilation and situations where existing automation failed to solve a significant percentage of side conditions. Additionally, in part due to design elements propagating upwards from Bedrock2 involving stack allocation and local variables, we found it easier to implement our array-processing lemmas as a separate component from Rupicola's existing mechanisms. While our success in grafting this machinery onto Rupicola's existing structure speaks to the extensibility of relational-compilation approaches, it reduced the availability and effectiveness of the automated style that Rupicola presently prefers. We speculate that designing future frameworks around manual, tactic-driven exploration of the compilation relation might facilitate relational compilation for a more varied domain of functional programs, without some of the challenges we faced in our effort.

3.6 Field Arithmetic from Fiat Cryptography

Specification technique: an optimizing compiler operating on PHOAS-encoded [Chlipala 2008] purely functional programs with a Bedrock2 backend specified using omnisemantics and separation-logic assertions with read-before-write aliasing.

We used Fiat Cryptography (FC) to generate implementations of basic modular-arithmetic operations used in elliptic-curve cryptography (Curve25519). Both the correctness proofs of the arithmetic-algorithm templates from Fiat Cryptography and the correctness proof of the partial evaluator [Gross et al. 2022] are used in the proof of our system, together guaranteeing functional correctness of the generated code, which is effectively straightline three-address code (FC IR), against very simple specifications in terms of integer arithmetic from the Coq standard library.

We chose to write and prove a compiler from FC IR to Bedrock2, and we are pleased to report that the independently expedient encoding choices in the two languages work well together. Specifically, the statement of the relevant compiler-correctness theorem applies to well-formed FC IR expressions that only use operators supported by Bedrock2 and RISC-V, and it guarantees that the generated Bedrock2 code can be executed with the expected memory-layout precondition and postcondition, which are expressed as a universally quantified omnisemantics judgment.

```
Definition shift_each (a : list Z) :=
  map (fun x => Z.shiftr x 1) a.
```

```
function(a : listZ) : listZ {
  out0 = Z_shiftr (List_nth a 0) 1
  out1 = Z_shiftr (List_nth a 1) 1
  return [out0, out1] }
```

Fig. 7. Elementwise shift in Gallina (left) and in Fiat Cryptography IR after simplification (right)

```
void shift_each(uint32_t *a, uint32_t *out) {
  out[0] = a[0] >> 1; out[1] = a[1] >> 1; }
```

Fig. 8. C code to shift each element in a 2-element list

Further, we express the memory-layout assertions using separation logic with non-separating conjunctions for function inputs: as expected for cryptographic field-arithmetic implementations, the code compiled from FC IR is always safe to call with overlapping inputs.

We contributed our FC-to-Bedrock2 compiler to FC. In addition to generating RISC-V code using the verified Bedrock2 compiler, the FC-derived Bedrock2 code can also be pretty-printed as C. This feature makes it usable as an alternative to the current unverified C backend of FC that directly pretty-prints FC IR as C, translating functional lists to C arrays on-the-fly. The new C backend provides formal guarantees about the memory-layout expectations of FC-generated C code.

3.6.1 Compiler Implementation. We will use a small code snippet to demonstrate how the compiler works. The Gallina code in Figure 7 shifts each element in a list. Proofs in FC normally interface with definitions like this one, where list lengths are not known. After specialization to a particular curve and passing through FC's pipeline, we will get an expression in FC IR that operates on fixed-length lists, such as the one on the right. Note that the `map` from the Gallina code has disappeared; this expression repeatedly indexes into the list instead.

Still, this IR looks pretty far from imperative code. To demonstrate the goal, Figure 8 shows a C version of the same operation, also specialized to a 2-element list. Ultimately, the compiler's goal for our example is to translate Figure 7 (right) into Bedrock2 code that approximates Figure 8.

The first hurdle the compiler must clear is that Bedrock2 has a concept of memory, while FC IR does not. The input to our final function should be pointers to arrays in memory, and the list literal that appears in FC IR is not directly expressible in Bedrock2. Here, the compiler takes advantage of certain specifics of FC's domain: lists used as function arguments have compile-time-known short lengths, contain integers, and are only indexed into with compile-time constants. Thus, it is okay to only load from memory at the very start of a function and only store at the very end. The compiler analyzes the type of the FC IR expression to find any lists in the arguments or return values. For lists in the arguments, the compiler creates a Bedrock2 expression that loads each element into a local variable. It passes this list of variables to the next stage of the compiler, which translates the main body of the function. If there are lists in the return type, the main-body translation returns a similar list of local variables for each one, allowing stores to be generated at the end.

The internal layers of the compiler can then associate program lists with compile-time lists of Bedrock2 variables. When we encounter an operation like `List_nth` in the IR, we can simply index into our compile-time list and extract the local variable we need. Translating internal expressions is relatively straightforward from there. The final code looks something like Figure 9.

Figure 10 shows the specification for the final Bedrock2 program. It requires that there is space in memory for the input and output at the given pointers. It ensures that the only memory modified is the memory at the output pointer, which now holds a shifted version of the input list. An important

```
func! (A_PTR, OUT_PTR) {
  x0 = load(4, A_PTR + 0); x1 = load(4, A_PTR + 4); x2 = x0 >> 1;
  x3 = x1 >> 1;            store4(x2, OUT_PTR + 0); store4(x3, OUT_PTR + 4) }
```

Fig. 9. Bedrock2 code for shifting 2-element lists

```
fnspec! "shift_each" pa pout / a x R ~> c,
{ requires t m :=
    m =*> array32 pa a ∧ length a = 2 ∧ m =*  array32 pout x * R ∧ length x = 2;
  ensures T M := exists out, M =* array32 pout out * R ∧
    length out = 2 ∧ T = t ∧ out = map (fun x => Z.shiftr x 1) a }.
```

Fig. 10. Bedrock2 specification for shifting a 2-element list

```
Definition shift_each_correct (shift_each : list Z -> list Z) :=
  forall a, shift_each a = map (fun x => Z.shiftr x 1) a.
```

Fig. 11. FC specification for shifting a list elementwise.

note about this specification is that it *does not require the input and output pointers be separate*. If they are the same, this function will perform an in-place shift.

One more detail of the compiler's implementation strategy is worth mentioning. The compiler is sound but not complete; it cannot compile all FC IR programs, e.g. when they use multidimensional lists. We formalize the exact precondition and predicate the compiler-correctness theorem on it. Coding of the compiler is then simplified by freeing us to return any correctly typed nonsense code when source programs use disallowed features, and the same cases are skipped in proofs.

3.6.2 Specifications and Composition. Final proofs of code from the compiler target Bedrock2-style specifications with separation-logic predicates, as shown in Figure 10. In order to connect the final code to its high-level mathematical behavior, we need to invoke: (1) the correctness proofs of FC templates; (2) the correctness proof of the pipeline that translates templates into FC IR; and (3) the correctness proof of the compiler itself.

The specifications for FC templates are expressed in very plain Coq language as predicates over pure functions; for example, the specification for the shift-each example would look something like Figure 11. When we apply the pipeline's correctness proof and compose it with the template proof, we get a statement that the function we get from interpreting the FC IR expression matches a specification like this one.

The main bridge we now need to cross is memory representation. We can write a one-time composition helper lemma for each general shape of operation (for example, one for unary operations on field elements and one for binary operations). These lemmas state: for any FC IR expression, if (a) the expression is accepted by the compiler's validity precondition and (b) there is a proof that the interpreted FC IR expression matches a certain Gallina function, then the Bedrock2 function fulfills a spec that integrates this Gallina function. For example, in Figure 10, the `map` function appears between statements about arrays and traces.

Finally, we apply the helper lemma when we specialize to a specific operation on a specific curve. Since the side conditions are formulaic, these proofs are completely automated with tactics. The output is a proof that the final Bedrock2 function matches a specific higher-level mathematical function, usually a field operation. This result has nothing specific to FC in it, making it straightforward to compose further.

Discussion. This implementation was a positive experience, validating our claims that our unified specification method (separation logic with limited use of non-separating conjunction) supports flexible use of multiple styles (direct Bedrock2 coding or targeting by this backend or Rupicola). One team member built the backend working 30-50% time over 9 months, which included dealing with rough edges in shared tactics that have since been improved. A later refactor, using about one person-month, improved code/proof reuse across compiled functions for different arithmetic primitives.

4 EVALUATION

We wanted to answer several questions about our approach.

(1) Do we succeed in generating machine code obeying conventions of embedded systems?
(2) Do we generate code with sufficient performance?
(3) How productive can developers be with this style?

First, we give some statistics on the scale of our case study. Recall that our development is structured mostly as Gallina functions, which are translated into low-level code in a variety of ways. We have 16 such functions: field-arithmetic operations, Montgomery-ladder Diffie-Hellman key exchange, ChaCha20, and the IP checksum. Through our several techniques, we compile these Gallina functions into Bedrock2 functions and combine them with handwritten code (the top-level loop, 5 trivial wrappers, Curve25519 `clamp`, the Ethernet driver, `memmove`, `memequal`, `memswap`, and `memconst`). Pretty-printing the Bedrock2 codebase as C yields a total of 2808 lines of code[5] across 37 functions.

Table 1. Memory usage of our generated RISC-V machine-code program

Allocation	**Bytes**
Stack	1,460
Static	1,616
Code	26,084
Total RW	3,076
Total	29,160

4.1 Suitability for Constrained Environments

The final system indeed meets standard restrictions for bare-metal microcontroller code. There is no heap allocation, and stack size is analyzed statically and proven to be disjoint from static allocations. The verified binary is position-independent, and we have proved termination of all functions but the main loop (which intentionally does not terminate). Our formal models and proofs do not cover side channels, but standard constant-time programming techniques are followed in both handwritten and generated code. For Rupicola, we specifically chose compilation rules that translate relevant source-language constructs to calls to constant-time functions (`felem_cswap` in Figure 6) written and proven using Bedrock2, exercising the same flexibility that Rupicola evaluation used for low-level performance engineering. The FC-to-Bedrock2 compiler always generates straight-line code with constant-time operations.

Table 1 gives the exact sizes of the three allocated memory segments in our statically linked executable, which add up to just over 3 KiB of read-write memory and under 32 KiB of read-only memory. As the FE310 (and many other common microcontrollers) executes code from external read-only memory (automatically fetching and caching instructions on-demand), there is substantial flexibility in the latter category. However, the 16 KiB of RAM on the FE310 would be easy to fill, for example by creating a new copy of the packet buffer each time its contents are modified.

4.2 Performance

We test our implementation using a simple Python client (on a laptop) that drives a network connection to the FE310 microcontroller, playing the client side of our cryptographic protocol to request that the garage door open or close. Table 2a compares our implementation's performance

[5]Counted using `cloc` after `clang-format` with default settings

Table 2. Performance results on the microcontroller

(a) End-to-end performance measurements of different implementations

Implementation	Time
Ours (compiled with Bedrock2)	0.47s
Ours (compiled with GCC)	0.12s
Ours (substituting BoringSSL)	0.12s
Test client initialization only	0.10s

(b) Performance comparison of X25519 implementations

Implementation	Cycles
This paper (with GCC)	4,144,983
van den Berg [2020]	5,389,988
BoringSSL	6,705,887

to two others, in terms of client-measured time to execute the network protocol. The first line reports our most high-assurance version: the complete software image compiled with Bedrock2, certified with one Coq theorem about a machine-code sequence. The next line reports on a moderately less high-assurance version, where we compile Bedrock2 code using GCC instead of the verified compiler. We see about a factor of four performance cost, which may be considered unsurprising, since the Bedrock2 compiler (which we did not modify) contains no standard compiler optimizations beside register allocation.

The third line of Table 2a compares with a version where we substitute our peculiarly generated X25519 code with the implementation from the popular BoringSSL library[6], which is used in e.g. the Chrome web browser. We were pleasantly surprised to see that we measured no performance degradation. The last line of the table shows the "best possible," relative to some parts of the experimental setup that we held fixed: the Python client that operates the demo system, which we here modified to only import and initialize the network and cryptography libraries but to skip interaction with the server. At least relative to that baseline, our embedded code compiled with GCC adds just 20% of client-observed execution time.

Those end-to-end results seemed promising, suggesting that we had achieved competitive performance on the cryptographic library, our primary new formally verified artifact in this project. We decided to zoom in on that aspect. Table 2b compares our implementation against two others, all compiled with GCC, with times measured using the microcontroller's cycle counter. BoringSSL is a natural baseline, though it has arguably not been optimized for RISC-V execution. In contrast, the master's thesis of van den Berg [2020], supervised by crypto-implementation expert Tanja Lange, was a credible effort to maximimze RISC-V performance of primitives from the NaCl crypto library[7]. Though we had been prepared to write this section of our paper apologetically, noting that performance costs are worth paying for increased assurance, we were surprised to find that we may have accidentally set a new performance record for this routine on RISC-V! Specifically, van den Berg [2020] runs an X25519 operation in about 80% the time of BoringSSL, and then our implementation runs in about 75% the time reported by van den Berg. Each reported measurement is the average of 10 measured runs after 2 discarded runs to warm up the instruction cache (which we confirmed to be adequate), with cycle counts extracted from the `mcycle` register using inline assembly. We should also mention that performance benchmarking is unusually straightforward in this domain as, for security reasons, routines are written to run in input-independent constant time, so any test inputs will do. We also benchmarked on a microcontroller with no operating system or other source of unpredictable scheduling.

[6] https://boringssl.googlesource.com/boringssl
[7] https://nacl.cr.yp.to/

Our conclusion is that it seems to be quite feasible to reach end-to-end performance parity using our methods, if more effort is expended on optimizations in parts of the existing verified artifacts that we did not focus on.

4.3 The Development Experience

Throughout section 3, we included discussion on the developer experience for different techniques, deeming some more worthy of near-future use than others. We can also say a bit more about the final code complexities of some representative modules.

Table 3. Lines of code of ChaCha20 (C) and Montgomery ladder (M)

Purpose	C	M
High-level algorithm	40	84
Main proofs	663	550
Rupicola lemmas & extensions	1282	132
Generic lemmas & automation	558	222

4.3.1 Effort in New Domains Versus Established Ones. To give an idea of the effort involved in verifying code in new domains versus established ones, we present Table 3, designed to give an idea of where we spent our development effort. The numbers represent lines of code dedicated to each purpose during the development of the ChaCha20 and Montgomery-ladder implementations respectively, beyond what the existing foundation of Rupicola and Fiat Cryptography provided. These numbers are not exact since work on both implementations involved some improvements to existing systems, but they are largely representative. Note that the greatest effort in terms of volume of code occurred in developing extensions to Rupicola for ChaCha20, in particular to support the patterns of loop unrolling and local-variable usage outlined earlier. Case-study code in other categories was much smaller, with the runner-ups being the network-protocol proof (650 lines for 55 lines of code), memmove (350 and 21 lines), and the network-driver improvements (107 lines for 14 lines of transmit path).

4.3.2 Effort of This Project. Table 4a lists each component that was newly developed for this paper, which technique it used, and its line count.

4.3.3 Size of the Code Base Including Dependencies. The source directory with our code and all dependencies contains over 3500 Coq files that amount to almost 400KLOC, but focusing on that measurement is misleading, because many of these files are from different research projects and not required for our project. To get a more representative count without resorting to manually classifying 3500 Coq files as on-topic or off-topic, we rely on the Coq command `Print Libraries` to list all files that the file containing our toplevel theorem transitively depends on, and then we only count the lines of code of these, which leads to the numbers in Table 4b.

4.4 Verification Time

The entire Coq development (proofs and code for our demo and dependencies) can be built in 20 minutes on a recent high-end computer (7950X3D averaging 5.2GHz on the fastest core). About 3 cores are utilized on average, so the same job takes longer on "server" hardware optimized for aggregate multicore performance at the expense of single-core performance. We measured 42 minutes on Xeon Gold 6268CL (2.8GHZ nominal) and 44 minutes on EPYC 7763 (3.2GHz nominal) as offered by popular cloud-based development environments. Incremental builds and interactive editing are much faster in most cases (often seconds), though outlier files such as the network-protocol proof and derivations of field-arithmetic and ChaCha20 code can each take a couple of minutes alone. 3.5 GB of memory is required.

Table 4. New and total lines of code of the project

(a) New lines of code of this paper

Component	Technique	LOC
Garage door app	Bedrock2	1142
Transmit for LAN9250	Bedrock2	134
Bedrock2 library functions	Bedrock2	418
ChaCha20	Rupicola	2591
Montgomery ladder	Bedrock2	374
Montgomery ladder	Rupicola	1062
Modular inversion	simpl, Rupicola	467
FC-to-Bedrock2 compiler	Generic Coq	7110
Invoking FC-to-Bedrock2 compiler	Generic Coq	804
Total		14102

(b) `Required` lines of code

Repository	LOC
Coq stdlib	85495
bedrock2	7402
bedrock2Examples	2759
compiler	14911
Coqprime	5650
coqutil	11424
Rewriter	30124
riscv	6077
Rupicola	8198
Fiat Crypto	83861
Total	255901

5 RELATED WORK

We are only aware of a few efforts to prove complete software-hardware stacks, and they have tended to involve software written in single languages and compiled in relatively conventional ways (albeit with verification applied to the compilers). Software in the pioneering CLI stack [Bevier et al. 1989] was written in Piton, whose programming model has similarities to Pascal and assembly languages. The Verisoft [Alkassar et al. 2008] and Bedrock2 [Erbsen et al. 2021] stacks rely on variants of C that allow more general memory-access patterns, producing code suitable for embedded systems. An odd one out in this picture is CakeML Silver [Lööw et al. 2019]: as the name suggests, the software in this project is written in an ML-style language, allowing the self-hosting compiler [Kumar et al. 2014] to be cross-compiled for the foundationally verified stack, but requiring use of a garbage collector. From this category, our past work [Erbsen et al. 2021] proved an end-to-end theorem about an application that receives network input, and none of them were used in such proofs covering both input and output with the network.

The VeriPhy framework [Bohrer et al. 2018] extends the CakeML work with a new connection to the KeYmaera X hybrid-system verifier [Fulton et al. 2015], allowing end-to-end verification of controller software for cyberphysical systems. This project demonstrated the important task of extending a verified stack upward with handling of application safety proofs in a quite-specialized domain. However, the foundational nature was weaker than in the follow-on work with CakeML and Silver: trusted aspects include a solver for real arithmetic, a translation between two variants of the HOL logic, some low-level systems code, and isolation across modules of low-level software.

Another purely software-based example is the DeepSpec web server and its applications [Zhang et al. 2021]. A simple web server was implemented in C and verified using the Verified Software Toolchain [Appel 2014]. The C code invokes system calls that are implemented by CertiKOS [Gu et al. 2016]. Moreover, some integration verification is carried out, with the application code proved against system-call specifications in a style used with CertiKOS. However, the proof of CertiKOS itself was not linked with the application proof, even at the level of C, let alone compiled machine code. The unifying specification technique of this work is interaction trees [Koh et al. 2019], which makes for an interesting contrast with our approach in this work (separation logic plus I/O traces logged in ghost state). All code in this DeepSpec work is implemented directly in C and verified, in contrast to our own work deriving C code from a variety of higher-level languages.

That last group of projects formulated their proofs *modularly*, with separate specifications for components, such that each can be proved once and then reused in other full systems without new proof. This appealing benefit comes at the cost of more sophisticated and labor-intensive verification, so quite a few projects have made other choices. Closest to our scope are Notary [Athalye et al. 2019] and Knox [Athalye et al. 2022], which consider hardware designs of embedded systems and the software running on top, doing unified symbolic execution (with SMT solvers) of the full stacks to establish correctness and security properties. However, many software stacks are complex enough on their own to raise the same tradeoff. Examples of systems applying symbolic execution across software layers include Serval and its predecessors in push-button verification of operating systems and more [Nelson et al. 2019, 2017; Sigurbjarnarson et al. 2016, 2018], plus Vigor [Zaostrovnykh et al. 2019] for network functions. These projects have demonstrated impressively low verification effort, but they provide reduced guarantees for their individual components, which often do not have their own specifications, so that assembly of new systems from old pieces may reveal new bugs (in addition to requiring full analysis from scratch of most source code). Additionally, none of these verifications are foundational.

Several formal-methods tools have seen significant use in verification of systems code stacks. The Dafny language has been applied as part of the Komodo [Ferraiuolo et al. 2017] trusted execution environment, the Ironclad [Hawblitzel et al. 2014] software-execution stack, and the IronFleet [Hawblitzel et al. 2017] distributed-systems platform. Less sweepingly, Coq has hosted functional programs implementing distributed systems in Verdi [Wilcox et al. 2015] and file systems in FSCQ and its variants [Chen et al. 2017, 2015; Ileri et al. 2018]. Such project architectures can support reasoning across significant spans of functionality, but they rely in the end on conventional compilation. As a result, (1) performance is often worse than with C-language counterparts, (2) compilers are unverified sources of bugs, and (3) deployed code requires garbage collection and is unsuited for resource-constrained embedded systems.

An interesting alternative approach is embodied by Goose, a tool built for the Perennial [Chajed et al. 2019] file-system verification. Goose translates handwritten Go code, which is importantly lower-level than in the projects just mentioned, to a core representation inside Coq, where proofs can be done using the Iris [Jung et al. 2015] framework. We give Perennial as just one example of verification of concurrent code, which our own case study does not include, allowing us to streamline the processes of both coding and proof.

Several projects have recognized the need for proof-generating compilation that produces code suitable for OS kernels and other low-level infrastructure. The seL4 team have developed Cogent [O'Connor et al. 2021] and the CertiKOS team DeepSEA [Sjöberg et al. 2019], which start from carefully designed source languages and add verified compilation to both C and functional programs embedded in theorem provers. A different take on the same challenge is the Low* [Protzenko et al. 2017] language, which is embedded inside the functional language F*, with constructs for explicit manipulation of memory. Our approach allows a system to be implemented in a high-level verification-oriented language, with no explicit memory management or imperative features, which makes for convenient proof of higher-level properties, nonetheless supporting programmer-guided generation of low-level code in different ways for different modules of a system.

Our application domain is cryptography, and some impressive past efforts have verified cryptographic libraries. Some work verifies off-the-shelf assembly code [Chen et al. 2014], while the Jasmin [Almeida et al. 2017] framework compiles programs written in a specialized low-level language. The F* ecosystem has HACL* [Zinzindohoué et al. 2017] and EverCrypt [Protzenko et al. 2020], whose proofs cover wide ranges of cryptographic functionality. The Verified Software Toolchain has examples connecting C programs to security specifications [Beringer et al. 2015; Ye et al. 2017]. However, all of these past projects entail writing low-level code manually, whether in

assembly, C, or a domain-specific language. In comparison to them, our new work is distinguished in allowing all cryptographic code to be written in a high-level language (Coq's pure functional programs) while avoiding any trust in compiler correctness. Additionally, to our knowledge, no past cryptographic library has been used as part of a broader *foundational* correctness proof that includes an application (let alone device drivers and all code needed for bare-metal execution).

That last proviso about foundational natures of proofs was important to distinguish from the tour de force of the Noise* project [Ho et al. 2022]. There we find a unified proof of a large family of cryptographic protocols, using F* and Low*, with significantly more complex cryptography than our case study features. Compile-time partial evaluation specializes the implementation to one protocol within the family, removing any lingering performance costs of generality. In common with several projects cited just above, there are proofs not just of functional correctness but also of cryptographic protocol security, which our case study in this paper does not provide. However, Noise* is much less foundational than our work, where its trusted base includes an SMT solver, the F* implementation, the Low* compiler, and a C compiler. Additionally, from the standpoint of different scopes of "end-to-end" proofs, while Low* pushes the scope upward to include cryptographic security, their verification also ends higher on the system stack, not including systems software like device drivers that is needed to support a full application.

Most closely related to our own effort are our past Coq projects that we build on. The core of our architecture is Bedrock2 [Erbsen et al. 2021], the low-level language to which almost all software modules are compiled. While Bedrock2's original lightbulb-controller case study included verified hardware, here we focus on more complex complete software images to run on a commercial microcontroller. We also apply the Rupicola [Pit-Claudel et al. 2022] compiler, which guides programmers in deriving C-like code from functional programs, again expanding upon its past applications, including compiling template-derived code and calling code compiled from other languages in Rupicola-generated code. Finally, we depend on Fiat Cryptography [Erbsen et al. 2019] to produce our cryptographic arithmetic, adding a verified backend targeting Bedrock2.

6 CONCLUSION

We structured our case study around compiling different parts of a pure functional program into C, combining it with handwritten C, and compiling it in a verified way down to machine code, with different ways of arriving at linkable formal proofs of the modules. We did avoid explicit low-level programming in most modules, sometimes with pretty much the ideal amount of per-module automation, sometimes by pushing programming decisions into proof scripts, which telegraph code choices through selections of proof rules. We showed that one style of specification (separation-logic specs that cite functional reference implementations) is sufficient as the interface glue between implementation styles. There remains much opportunity to streamline the experience, with Coq performance bottlenecks being our most common stumbling block, often leading to more verbose program derivations than we already know how to implement in standalone compilers. Nonetheless, the current state of the art was sufficient to derive the code for a bare-metal cryptographic server that runs in a physical prototype.

ARTIFACT-AVAILABILITY STATEMENT

An open-source artifact [Erbsen et al. 2024] for this paper was evaluated and is freely available.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grants CNS-2130671, CCF-2217064, and CCF-2313023; the National Science Foundation Expedition on the Science of Deep

Specification (award CCF-1521584); the National Science Foundation Graduate Research Fellowship Program under Grant No. 2141064; and gifts from Amazon, Google, and the Tezos Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. 2008. The Verisoft Approach to Systems Verification. In *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08) (LNCS, Vol. 5295)*, Natarajan Shankar and Jim Woodcock (Eds.). Springer, 209–224. https://doi.org/10.1007/978-3-540-87873-5_18

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1807–1823. https://doi.org/10.1145/3133956.3134078

Andrew W. Appel. 2014. *Program Logics - for Certified Compilers*. Cambridge University Press. https://www.cs.princeton.edu/~appel/papers/plcc.pdf

Anish Athalye, Adam Belay, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2019. Notary: A Device for Secure Transaction Approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 97–113. https://doi.org/10.1145/3341301.3359661

Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying Hardware Security Modules with Information-Preserving Refinement. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 503–519. https://www.usenix.org/conference/osdi22/presentation/athalye

Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 207–221.

William R. Bevier, Warren A. Hunt, Jr., J. Strother Moore, and William D. Young. 1989. An approach to systems verification. *J. Autom. Reasoning* (1989), 411–428. https://doi.org/10.1007/BF00243131

Rose Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. 2018. VeriPhy: Verified Controller Executables from Verified Cyber-Physical System Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 617–630. https://doi.org/10.1145/3192366.3192406

Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Pratap Singh, Andrew Wright, and Adam Chlipala. 2023. Flexible Instruction-Set Semantics via Abstract Monads (Experience Report). In *ICFP'23: Proceedings of the 28th ACM SIGPLAN International Conference on Functional Programming* (Seattle, WA, USA). http://adam.chlipala.net/papers/RiscvICFP23/

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying Concurrent, Crash-Safe Systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 243–258. https://doi.org/10.1145/3341301.3359632

Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Trans. Program. Lang. Syst.* 45, 1, Article 5 (mar 2023), 43 pages. https://doi.org/10.1145/3579834

Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 270–286. https://doi.org/10.1145/3132747.3132776

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. 2014. Verifying Curve25519 Software. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14*. ACM, 299–309. http://cryptojedi.org/papers/#verify25519 Document ID: 55ab8668ce87d857c02a5b2d56d7da38.

Adam Chlipala. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 143–156. http://adam.chlipala.net/papers/PhoasICFP08/ 10.1145/1411204.1411226.

Andres Erbsen. 2022. *Foundational Integration Verification of Diverse Software and Hardware Components*. Ph. D. Dissertation. Massachusetts Institute of Technology. http://adam.chlipala.net/theses/andreser.pdf

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 604–619. https://doi.org/10.1145/3453483.3454065

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. *2019 IEEE Symposium on Security and Privacy (SP)* (May 2019). https://doi.org/10.1109/sp.2019.00005

Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Samuel Gruetter, Adam Chlipala, and Clément Pit-Claudel. 2024. *Foundational Integration Verification of a Cryptographic Server*. https://doi.org/10.5281/zenodo.10807084

Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 287–305. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/10/komodo.pdf 10.1145/3132747.3132782.

Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. 2015. KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems. In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 527–538.

Jason Gross, Andres Erbsen, Jade Philipoom, Miraya Poddar-Agrawal, and Adam Chlipala. 2022. Accelerating Verified-Compiler Development with a Verified Rewriting Engine. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPICS.ITP.2022.17

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2017. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. *Commun. ACM* 60, 7 (June 2017), 83–92. https://doi.org/10.1145/3068608

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 165–181. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. 2022. Noise*: A Library of Verified High-Performance Secure Channel Protocol Implementations. In *2022 IEEE Symposium on Security and Privacy (SP)*. 107–124. https://doi.org/10.1109/SP46214.2022.9833621

Mirai Ikebuchi, Andres Erbsen, and Adam Chlipala. 2022. Certifying derivation of state machines from coroutines. *Proc. ACM Program. Lang.* 6, POPL, Article 24 (jan 2022), 31 pages. https://doi.org/10.1145/3498685

Atalay Ileri, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2018. Proving Confidentiality in a File System Using DISKSEC. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 323–338.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. https://doi.org/10.1145/2676726.2676980

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 234–248. https://doi.org/10.1145/3293880.3294106

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://www.cse.chalmers.se/~myreen/popl14.pdf 10.1145/2535838.2535841.

Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. 2019. Verified compilation on a verified processor. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1041–1053. https://cakeml.org/pldi19.pdf 10.1145/3314221.3314622.

Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 225–242. https://doi.org/10.1145/3341301.3359641

Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 252–269. https://doi.org/10.1145/3132747.3132748

Liam O'Connor, Zilin Chen, Christine Rizkallah, Vincent Jackson, Sidney Amani, Gerwin Klein, Toby Murray, Thomas Sewell, and Gabriele Keller. 2021. Cogent: uniqueness types and certifying compilation. *Journal of Functional Programming* 31 (2021), 25. https://doi.org/10.1017/S095679682100023X

Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. 2022. Relational Compilation for Performance-Critical Applications: Extensible Proof-Producing Translation of Functional Models into Low-Level Code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 918–933. https://doi.org/10.1145/3519939.3523706

Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12167)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, 119–137. https://doi.org/10.1007/978-3-030-51054-1_7

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-Beguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *2020 IEEE Symposium on Security and Privacy (SP)*. 983–1002. https://doi.org/10.1109/SP40000.2020.00114

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *Proc. ACM Program. Lang.* 1, ICFP, Article 17 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110261

Peter Schwabe, Benoît Viguier, Timmy Weerwag, and Freek Wiedijk. 2021. A Coq proof of the correctness of X25519 in TweetNaCl. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–16. https://doi.org/10.1109/CSF51468.2021.00023

Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 1–16. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/sigurbjarnarson

Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. 2018. Nickel: A Framework for Design and Verification of Information Flow Control Systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 287–305. https://unsat.cs.washington.edu/papers/sigurbjarnarson-nickel.pdf

Vilhelm Sjöberg, Yuyang Sang, Shu-chun Weng, and Zhong Shao. 2019. DeepSEA: A Language for Certified System Software. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 136 (oct 2019), 27 pages. https://doi.org/10.1145/3360562

Stefan van den Berg. 2020. RISC-V implementation of the NaCl-library. Master's thesis. https://pure.tue.nl/ws/portalfiles/portal/169647601/Berg_S._ES_CSE.pdf

Andrew Waterman and Krste Asanovic. 2019. The RISC-V Instruction Set Manual. https://riscv.org/specifications/.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. *SIGPLAN Not.* 50, 6 (June 2015), 357–368. https://doi.org/10.1145/2813885.2737958

Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. 2017. Verified Correctness and Security of MbedTLS HMAC-DRBG. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2007–2020. https://doi.org/10.1145/3133956.3133974

Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying Software Network Functions with No Verification Expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 275–290. https://vigor-nf.github.io/vigor-paper.pdf 10.1145/3341301.3359647.

Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1789–1806. https://doi.org/10.1145/3133956.3134043

Received 2023-11-16; accepted 2024-03-31