Galápagos: Developing Verified Low-Level Cryptography on Heterogeneous Hardware

Yi Zhou
Carnegie Mellon University
Pittsburgh, PA, United States
yizhou5@andrew.cmu.edu

Sydney Gibson
Carnegie Mellon University
Pittsburgh, PA, United States
sydneyg@andrew.cmu.edu

Sarah Cai
Databricks
San Francisco, CA, United States
sarah.cai@gmail.com

Menucha Winchell
University of California, Berkeley
Berkeley, CA, United States
menuchawinchell@gmail.com

Bryan Parno
Carnegie Mellon University
Pittsburgh, PA, United States
parno@cmu.edu

ABSTRACT

The proliferation of new hardware designs makes it difficult to produce high-performance cryptographic implementations tailored at the assembly level to each platform, let alone to prove such implementations correct. Hence we introduce Galápagos, an extensible framework designed to reduce the effort of verifying cryptographic implementations across different ISAs.

In Galápagos, a developer proves their high-level implementation strategy correct once and then bundles both strategy and proof into an abstract module. The module can then be instantiated and connected to each platform-specific implementation. Galápagos facilitates this connection by generically raising the abstraction of the targeted platforms, and via a collection of new verified libraries and tool improvements to help automate the proof process.

We validate Galápagos via multiple verified cryptographic implementations across three starkly different platforms: a 256-bit special-purpose accelerator, a 16-bit minimal ISA (the MSP430), and a standard 32-bit RISC-V CPU. Our case studies are derived from a real-world use case, the OpenTitan security chip, which is deploying our verified cryptographic code at scale.

CCS CONCEPTS

• **Software and its engineering → Software verification**.

KEYWORDS

cryptographic implementation; assembly code; program verification; heterogeneous hardware

ACM Reference Format:
Yi Zhou, Sydney Gibson, Sarah Cai, Menucha Winchell, and Bryan Parno. 2023. Galápagos: Developing Verified Low-Level Cryptography on Heterogeneous Hardware. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616603

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3616603

1 INTRODUCTION

As Moore's law slows, we have seen an explosion of new, custom hardware designs that aim to increase performance and/or reduce power consumption relative to general-purpose processors [33, 35, 36, 41]. In our IoT-entranced world, these devices are inevitably connected to the Internet, and hence require cryptographic implementations for tasks like checking firmware integrity or establishing secure connections to remote servers. Such tasks place the cryptographic implementation on the system's critical path, making high performance crucial.

Historically, cryptographic providers such as OpenSSL [47] have met these performance demands via hand-written assembly code that utilizes platform-specific optimizations (e.g. NEON [6] or AES-NI [26]), capturing performance gains missed by generic compilers. Emerging heterogeneous platforms reinforce this trend, since compilers for them (including one of our case studies) may not be developed until long after the platforms are deployed, making hand-crafted low-level code a necessity.

Unfortunately, manually writing such low-level code invites vulnerabilities; e.g., OpenSSL has reported 33 CVEs since 2021 [46], of which 29 are memory safety or function correctness bugs. Formal software verification can statically prove an implementation free of entire classes of vulnerabilities, but prior work in this area is ill suited to a world of heterogeneous hardware (§6).

When supporting heterogeneous platforms, *verification cost* and *specialization-based performance* are at odds. A large swath of work [5, 9, 21, 53, 59, 65, 68] verifies high-level source code and then assumes a standard compiler produces correct assembly without introducing vulnerabilities. This approach reduces verification costs, but it sacrifices specialization-based performance gains [11]; it is also infeasible for platforms that lack a compiler. Other work directly targets assembly implementations [2, 3, 11, 12, 14, 24, 54, 55, 61]. This approach retains performance but targets only specific platforms. Hence the effort to verify a cryptographic algorithm (say, ECDSA [32]) grows linearly with the number of platforms targeted.

Our Approach. We present the extensible Galápagos[1] framework, which reconciles the need for low-cost verification with the

[1]The Galápagos finch and tortoise species are famous for adapting their bodies to the different environments on each of the Galápagos Islands. In the same vein, the Galápagos framework adapts cryptographic algorithms to the specifics of each supported hardware model.

performance gains from specialization in the multi-platform setting. Taking a cross-platform view emphasizes the importance of creating reusable abstractions across platforms, amortizing development costs. Galápagos supports such abstractions by allowing the developer to write high-level implementations and proofs that are *parameterized* by an abstract machine model, making them machine-independent. Galápagos also generates a common high-level interface for hardware ISAs, making it easier for the developer to connect platform-specific reasoning to the machine-independent proofs. These two forms of abstraction significantly reduce the developer's hardware-specific proof work, without compromising the run-time performance of their code.

Abstract Implementation. A Galápagos developer initially writes an abstract implementation that captures their machine-independent decisions and proves them correct. They use as many named variables as they wish (unconstrained by finite registers), interact with immutable sequences of structured data (rather than byte-level memory accesses), and can thus focus on proving the algorithm's mathematical correctness. For example, the developer might decide to implement the Cooley-Tukey (CT) algorithm (Algorithm 2) to realize the number theoretic transform (NTT). The correctness of CT is justified by the properties of polynomial rings, which can be proven independent of any specific platform.

The abstract implementation is bundled into a *functor* using support we added to Dafny (§3.1). A functor is a special type of module (a collection of types, functions, and proofs) that takes one or more modules as arguments and produces a new module. In our case, the abstract-implementation functor is parameterized by an abstract machine module that provides generic word-size operations, which makes the functor *reusable* across architectures. For instance, the classic Montgomery multiplication algorithm (Algorithm 1) is described in terms of some unspecified radix (word size), and the core operations (e.g., addition and multiplication), the various iteration counts, and even the pre-computed constants all depend on the radix. Nonetheless, the algorithm can be proven generically correct given an abstract machine model.

Platform-specific Instantiation. To target a new platform, as with prior work, the programmer must obtain (or write) a specification that defines the semantics of the hardware's ISA. For example, they might define a machine module with 256-bit words that supports addition and multiplication via hardware-specific instructions. They can then apply the abstract implementation's functor to this machine-specific module to instantiate a machine-specific module (containing a machine-specific algorithm and corresponding proofs). Note that this instantiated module is *obtained for free*, and it is now committed to the 256-bit word size.

Assembly Implementation. In the final step, the developer must show that an assembly implementation is working as described by the machine-specific algorithm in the instantiated module above. The assembly can be hand-written, produced by a compiler, or any combination thereof. Regardless, the developer must prove that each assembly routine realizes an algorithmic step (typically a fairly straightforward process). Crucially, however, they do not need complex proofs showing why those algorithmic steps are correct. Those proofs come for free from the instantiated module!

However, the instantiated module still operates over a high-level structured memory, whereas a hardware-level ISA typically operates over bytes. To manage this complexity, Galápagos supplies tools to *automatically* raise the level of abstraction for each platform. Specifically, Galápagos provides a functor-based, verified abstraction layer that translates a machine's low-level byte-oriented memory interface into a memory with a structured heap and stack.

Tooling and Library. To help with proof reuse and automation, Galápagos includes several improvements to the Dafny language as well as its first standard library.

Functors for Dafny. Creating and managing abstractions is critical for Galápagos. Hence, we introduced verified, ML-style functors to Dafny. This required adapting higher-order functional concepts to Dafny's imperative, first-order design.

Algebra Solver. Non-linear arithmetic is endemic to cryptographic algorithms. However, the state-of-the-art SMT solvers, which tools such as Dafny rely on, struggle to reliably handle non-linear reasoning [22, 30]. Prior work has shown the effectiveness of algebra solvers in the Coq interactive theorem prover [61]. We added similar support to the Dafny automated theorem prover, resulting in more concise proofs.

Standard Library. We developed the first standard library for Dafny (now distributed and maintained by the Dafny engineering team at Amazon) with over 5,800 LoC, 80 definitions, and 381 lemmas providing extensive verified facilities for reasoning about collections (e.g., sequences of bytes), translations between different ways of representing large integers in word-sized chunks, and a comprehensive collection of properties about non-linear arithmetic.

Case Studies. We base our validation of Galápagos (§4) on a real-world use case: the OpenTitan security chip [49]. Designed by partners including lowRISC and Google, OpenTitan is an open source TPM-like [60] chip that can provide a hardware root of trust for a wide variety of devices and applications. At the heart of OpenTitan's security architecture is a secure boot process [25, 50] that loads and executes properly signed code only. The code implementing OpenTitan's secure boot (including the cryptographic routines) is baked into the chip's ROM, meaning that any flaws must be addressed by physically recalling the flawed chips, printing a new multi-million-dollar hardware mask, and then fabricating and distributing new chips.

Further complicating the story, OpenTitan includes both a 32-bit RISC-V [57, 63] main core and a custom 256-bit big-number accelerator (dubbed the OTBN), and for extra resiliency, OpenTitan aims to support secure booting with and without the OTBN enabled. Hence, in our case studies, we have used Galápagos to produce fully verified implementations of OpenTitan's existing RSA-3072 signature verification routines for both RISC-V and OTBN. Our verified code has been burnt into the mask ROM currently in use for fabricating OpenTitan chips, the first instance, to our knowledge, of formally verified cryptographic code baked into hardware at scale.

To further validate Galápagos's ability to support heterogeneous hardware, we developed (in less than a week) an implementation for yet another architecture, the MSP430, a tiny 16-bit ISA with only 27 instructions, developed by TI for low-power embedded devices. We intentionally avoided ARM and x86 since they are quite standard and well studied in prior work [2, 3, 11, 12, 14, 54, 55].

To validate Galápagos's algorithmic generality, and to support OpenTitan's ongoing exploration of possible post-quantum algorithms, we have also produced verified implementations, for the

```
procedure times4()
  requires a0 < 100;
  reads a0; modifies a1, a2;
  ensures a2 == 4 * a0;
{
  add32(a1, a0, a0); // a1 <- a0 + a0
  add32(a2, a1, a1); // a2 <- a1 + a1
}
```

Figure 1: Sample RISC-V Code in Vale.

MSP430, RISC-V, and OTBN chips, of the post-quantum Falcon signature algorithm [23] recently standardized by NIST. Falcon's mathematical underpinnings differ starkly from RSA's. Falcon operates over lattices, and at the heart of our Falcon implementations is an NTT functor, parameterized by a polynomial ring, that can be used independently for other post-quantum algorithms. To our knowledge, these are the first verified Falcon implementations.

Our evaluation (§5) finds that Galápagos reduces the effort to define a new ISA by 30-50%, and the proof burden for target-specific implementations by 30-60%. Further, Galápagos's approach produces implementations with speed comparable to (and in some cases faster than) our unverified reference implementations.

Altogether, our case studies consist of approximately 36K lines of specification, code, and proofs, which, along with our tool improvements, are available online as open source [66].

Limitations. Galápagos still requires the developer to produce low-level implementations of their algorithms; for scenarios where compilers exist and performance is not essential, other approaches may require less developer effort. Our case studies focus on signature verification, where side channels are irrelevant, so Galápagos concentrates on functional correctness; standard extensions from prior work [11] could support reasoning about side channels. Like any verification effort, the soundness of our results depends on the correctness of our specifications (both of the cryptography and the machine semantics) and of our verification tool (Dafny).

Contributions. In summary, this research:

- Presents the Galápagos framework, which reduces developer effort for cross-platform cryptographic implementations.
- Introduces functor support into an SMT-based automated theorem prover, and shows how to use functors to abstract algorithms and heterogeneous platforms.
- Evaluates the reuse enabled by Galápagos on six verified implementations covering classical and post-quantum cryptographic algorithms and three disparate hardware platforms.
- Contributes a new verified Dafny standard library, now upstreamed, to facilitate future verification efforts.
- Produces the first formally verified cryptographic routines baked into hardware for large scale deployment.

2 BACKGROUND

Vale. Galápagos builds atop the Vale framework [11], which supports the verification of low-level, high-performance code. Figure 1 shows a sample Vale procedure that quadruples its input. The procedure's signature declares that it reads from register `a0` and modifies registers `a1` and `a2`. It also claims that if the input satisfies its precondition (the `requires` clause), then the output in `a2` will satisfy the postcondition (the `ensures` clause). It makes two procedure calls, which here correspond to individual assembly instructions.

```
datatype reg32_t =
  | A0
  | A1
  | ...

type mem_t = map<int, uint8>

datatype state = state(
  regs: regs_t, // 32-bit registers
  mem: mem_t, // Linear memory
  ok: bool) // Not crashed

// base integer instruction set, 32-bit
datatype Ins32 =
  | RV_ADD (rd: reg32_t, rs1: reg32_t, rs2: reg32_t)
  | RV_LW (rd: reg32_t, rs1: reg32_t, imm12: uint32)
  | ...

predicate eval_ins32(ins: Ins32, s: state, r: state) {
  match ins
    case RV_LW(rd, rs, imm) ⇒
    // load word from s.mem[rs + imm], set ok to false if unaligned
    ...
}

predicate eval_code(c: code, s: state, r: state) {
  match c
    case Ins32(ins) ⇒ eval_ins32(ins, s, r)
    case Block(block) ⇒ eval_block(block, s, r)
    ...
```

Figure 2: Sample RISC-V Semantics in Dafny.

Vale discharges proof obligations (e.g., that the preconditions imply the postconditions) by embedding the implementation code in a backend verifier (in our case, Dafny) which reasons about the implementation using a model of the target machine's hardware semantics. The verifier produces mathematical formulas and checks their validity with an SMT solver (in our case, Z3 [17]).

Thus, Vale proofs of correctness require a formal semantics for the underlying hardware. These may come from the hardware manufacturer (e.g., from ARM [56]), from prior academic work [7, 16], or the developer can write their own. Figure 2 shows a simplified sample of such a definition. It declares that the machine's state consists of a collection of named 32-bit registers, a memory that maps integer addresses to bytes, and an `ok` flag that indicates whether code has executed successfully without crashing. The `eval_code` predicate defines the semantics, i.e., it dictates how the execution of code `c` causes the machine to transition from state `s` to state `r`.

To aid proofs about their implementation, the Vale developer typically writes and proves additional lemmas directly in the backend verifier and invokes them from Vale.

Dafny Abstract Modules. Galápagos exploits proof reuse, which standard Dafny supports (to a degree) through abstract modules. An abstract module declares an interface, which can be implemented by concrete modules. Dafny generates verification conditions that ensure the concrete module adheres to the interface.

Consider the example in Figure 3. The abstract module `ring` declares an `elem` type and functions over it. The `int_ring` module `refines` the interface by declaring that `elem` has type `int` and providing bodies to the functions. Importantly, `add`'s body must satisfy the idempotency property specified in the abstract module.

Dafny also allows an abstract module to `import` other abstract modules, allowing access to their contents. Continuing with our example, suppose we want to implement a forward NTT generically over any ring. In `FNTT`, we can use the syntax `import R: ring` to use an unspecified module `R` that promises to implement the `ring`

```
abstract module ring {
  type elem
  function unit(): elem
  function add(a: elem, b: elem): (c: elem)
    ensures b == unit() ⟹c == a // Specifies idempotency
}
module int_ring refines ring {
  type elem = int
  function unit(): elem { 0 }
  // Success: idempotency maintained
  function add(a: elem, b: elem): elem { a + b }
  // Error: idempotency violated
  // function add(a: elem, b: elem): elem { b - a }
}
abstract module FNTT {
  import R: ring
  function double(a: R.elem): R.elem { R.add(a, a) }
  // Other generic implementations elided
}
abstract module INTT {
  import R: ring /* Generic implementations elided */
}
abstract module poly_mul {
  import F: FNTT
  import I: INTT
  // Problem: cannot express that F.R is the same module as I.R
  function problematic(a: F.R.elem, b: I.R.elem): F.R.elem {
    F.R.add(a, b) // Error: this does not type check
  }
}
```

Figure 3: Example Abstract Modules in Standard Dafny.

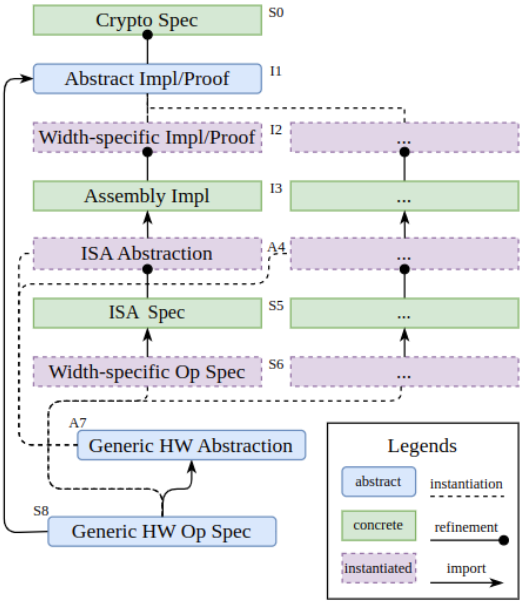

Figure 4: Galápagos Overview. An abstract implementation and proof (I1) parameterized by generic machine operations (S8) is proven to refine a crypto spec (S0). An assembly implementation (I3) is proven to refine a width-specific instance (I2) of the abstract impl (I1). The assembly implementation (I3) is written on top of an automatically generated instance (A4) of a higher-level hardware interface (A7), which is proven sound against the low-level ISA spec (S5). The ISA spec, in turn, is defined using an instance (S6) of the generic machine operations (S8). Given S0, I1 is written once; I3 and S5 are written once per-platform; and Galápagos provides I2, A4, S6, A7, and S8. Figure 12 shows how our case studies apply this workflow.

interface. Now we can use functions in `R` to perform more complex operations without assuming a particular implementation of `R.add`.

However, Dafny's basic module system falls short in a subtle but important case. Suppose now we want to implement an inverse NTT, and then use the two NTT modules to implement polynomial multiplication, all generically over some ring. The issue arises with `poly_mul`, where Dafny has no way to specify that the imported modules `F` and `I` are parameterized by the same underlying ring.

3 THE GALÁPAGOS FRAMEWORK

Galápagos is an extensible framework for developing high-performance cryptographic implementations on different platforms. As shown in Figure 4, the developer proves an abstract implementation (§3.2) correct once and then reuses it across different platforms. For each platform ISA, Galápagos automatically generates a proven-correct, higher-level interface (§3.3). The concrete assembly implementation (§3.4) can thus be written on top of this interface, allowing easier access to the proofs provided by the abstract implementation.

To support an existing crypto primitive on a new platform, the developer supplies a new ISA specification and a corresponding assembly implementation. To support a new cryptographic primitive on existing platforms, the developer adds a new cryptographic specification, along with corresponding assembly implementations. The remainder (shown in purple) comes automatically from Galápagos.

To provide the abstractions needed to achieve code reuse and amortize development costs, Galápagos relies on our introduction of functors to Dafny (§3.1). Proof automation is further aided by new solver support (§3.5) and standard libraries (§3.6) that we added.

3.1 Adding Functor Support to Dafny

Galápagos relies on abstraction to reduce developer effort. Dafny's existing module system was too limited for Galápagos (§2), so we expanded its expressivity by introducing ML-style functors [20].

Functors are functions from modules to modules. In our implementation, a functor is a module that takes other modules as arguments (each argument is given a type defined by an abstract module), and the code and proofs in the functor are written in terms of the module arguments. The developer can instantiate the functor by applying it to concrete modules that refine the formal arguments' types. A functor thus allows a collection of code and proofs to be reused when instantiated with different module arguments.

Using functors, we can now successfully implement the polynomial multiplication example from §2. As shown in Figure 5, `FNTT` is now a functor that takes a module `R` of type `ring` as an argument and returns an instantiation of the FNTT code and proofs specific to that concrete argument. Applying `FNTT` to a different ring module produces a different concrete instantiation. The crucial benefit of using functors (as opposed to Dafny's existing module system) is that when two functors are applied to the *same* argument (e.g., the `ring` module in `poly_mul`), we can successfully unify the types coming from the two different instantiated modules. Below, we expand on our functor design choices using Dreyer's terminology [20].

```
abstract module ring { type elem /* details elided */ }
abstract module FNTT(R: ring) { /* details elided */ }
abstract module INTT(R: ring) { /* details elided */ }
abstract module poly_mul(R: ring, F: FNTT(R), I: INTT(R)) {
   // Functions in F and I can interop since R is the same in both
}
```

Figure 5: Functor Example in Galápagos.

Applicative. Our functors are applicative, meaning that applying the same functor to the same argument(s) in two different contexts still produces the same concrete module. This is crucial for unifying types in examples like Figure 5. Our design contrasts with SML's generative functors, where each application generates a fresh copy of types, even with the same argument module(s). For example, in `A = FNTT(IntRing)` and `B = FNTT(IntRing)`, `A.elem` and `B.elem` will not be of the same type with generative functors.

Second-Class, First-Order. Similar to most ML dialects, our functors are second class, meaning the module system exist in a different plane from ordinary functions and types. Specifically, a module cannot be passed to or returned from ordinary functions, nor can it be stored in datatypes. Our functors are close to being first-order, since they cannot be partially applied, but they can be parameterized by other functors, which is a higher-order property.

Proof Obligations. Unlike most other functor-supporting languages such as OCaml or ML, Dafny's types and methods come with verification obligations. Hence, when extending Dafny to support functors, we had to carefully ensure that the proof of a functor's correctness relies only on the properties promised by the abstract module "types" of its formal parameters, not any details of the concrete instantiations. In exchange, we gain verification efficiency: we need only verify the abstract implementation once; i.e., no additional verification work is required when instantiating the functor with concrete module arguments, since those arguments have already been proven to refine the corresponding abstract modules.

3.2 Writing an Abstract Implementation

A key aspect of the Galápagos framework is that the developer initially writes an abstract implementation of their desired cryptographic primitive. This implementation captures their algorithmic decisions and optimizations. Since it is written against a generic, high-level machine model, proving these decisions and optimizations is much simpler than it would be for a concrete implementation cluttered with hardware-specific details like finite registers, byte-level memory access, etc. Once the developer instantiates the generic machine module for a concrete hardware platform, Galápagos provides a hardware-specific version of the correctness proofs. To illustrate this process, we first introduce the generic machine model and then show how the developer uses it to write their abstract implementation and prove it correct.

Generic Machine Operations. As shown in Figure 6, the Galápagos generic machine model is provided as an abstract module in Dafny. An *abstract* module (§2) omits implementations, so that other modules can provide those details by *refining* the abstract module in different ways. For instance, in the generic machine, `uint` represents the architecture's word size, but it is defined in terms of the upper bound `BASE()`, which deliberately omits a definition.

Within this module, Galápagos then provides various common hardware operations, including arithmetic operations, bit shifts, etc.

```
abstract module generic_machine_ops {
  // Symbolic upper bound on word size
  // Concrete instantiations must satisfy the ensures clauses
  function BASE(): (v: nat)
    ensures (v > 1)
    ensures (v % 2 == 0)

  // Defines an unsigned integer type upper-bounded by BASE()
  type uint = i: int | 0 ≤ i < BASE()

  /* Generic operations obtained "for free" by concrete
   * instantiations when they define BASE() */

  // Word-sized addition with carry
  function addc(x: uint, y: uint, cin: uint1): (uint, uint1) {
    var sum := x + y + cin;
    // Handle possible overflow
    var sum' := if sum < BASE() then sum else sum - BASE();
    var cout := if sum < BASE() then 0 else 1;
    (sum', cout)
  }

  // Extract the most-significant bit
  function msb(x: uint): uint1 {
    if x ≥ BASE()/2 then 1 else 0
  }

  // Interpret a sequence of uint as a natural number
  function to_nat(xs: seq<uint>): nat {
    // Details elided
  }

  // More operations elided
}
```

Figure 6: Snippet of Galápagos Generic Machine Operations in Dafny. Operations are defined with respect to an unknown word size `uint`; e.g., addition with carry wraps when the sum overflows.

These are defined in terms of `uint` words, without any knowledge of what the actual value of `uint` will be, other than the information from the `ensures` clauses, i.e., that `BASE()` will be even and larger than 1, which is convenient, for example, when defining `msb`.

To target a new platform, the Galápagos developer starts with a concrete module that refines the generic module above by filling in the missing definitions; for example, here is an excerpt of the definition for the 16-bit operations.

```
module bw16_ops refines generic_machine_ops {
  function BASE(): (v: nat) { 0x10000 }
  // addc, msb and to_nat are obtained for free!
}
```

Dafny checks that the refinement is valid (e.g., that the definition of `BASE()` is even and greater than 1 in this case) and then automatically fills in concretized versions of the abstract operations. In other words, we can now invoke `bw16_ops.addc` to talk about add-with-carry over 16-bit words.

Abstract Implementation. With Galápagos, a developer aims to capture the essence of their implementation strategy while abstracting away the complexities of a low-level executable. This makes proofs of correctness far simpler. The abstraction of implementation details takes several forms.

First, the developer can use an unlimited number of named variables, rather than worry about finite registers. Second, rather than reason about byte-level memory operations, they instead write their implementation by reading and updating immutable sequences of

```
abstract module generic_big_add_impl(ops: generic_machine_ops)
{
  function big_add(xs: seq<uint>, ys: seq<uint>, cin: uint1)
      : (seq<uint>, uint1)
    requires |xs| == |ys|
  {
    var len := |xs|;
    if len == 0 then
      ([], cin)
    else
      var (zs, cin') := big_add(xs[..len-1], ys[..len-1], cin);
      var (z, cout) := ops.addc(x[len-1], y[len-1], cin');
      (zs + [z], cout)
  }

  lemma big_add_correct(xs: seq<uint>, ys: seq<uint>,
      zs: seq<uint>, cout: uint1)
    requires |xs| == |ys|
    requires (zs, cout) == big_add(xs, ys, 0)
    ensures |zs| == |xs|
    ensures ops.to_nat(xs) + ops.to_nat(ys) ==
      ops.to_nat(zs) + cout * pow(BASE(), |xs|)
  {
    // Proof code here
  }
}

module some_client {
  import bw16_big_add = generic_big_add_impl(bw16_ops)
  // Free to use the 16-bit version of big_add and big_add_correct
}
```

Figure 7: Generic Dafny Multi-Word Addition Code & Proof

structured data (e.g., word-sized values). When a sequence is updated, it produces a copy of the original sequence with the corresponding element changed (similar in spirit to copy-on-write files). Hence every sequence is unique and unchanging, making reasoning far simpler since it, among other benefits, eliminates any aliasing concerns. Finally, the developer writes their implementation using the operations from the generic machine model (Figure 6).

To illustrate this process, Figure 7 shows an example of an abstract implementation of multi-word addition. Algorithms like RSA operate over large integers that cannot fit into a single machine word and must instead be represented by a sequence of words (or "limbs") stored in memory. In the example, when we define addition (`big_add`) over large integers, instead of explicitly referencing the memory, `xs` and `ys` are each represented using an immutable sequence of machine-words. Because sequences are ordinary values (just like integers), Dafny can trivially see that modifications to `xs` have no effect on `ys` (and vice versa), whereas a low-level implementation would have to worry about potential pointer aliasing. The implementation defines multi-word addition recursively, using variables like `len` and `z` to represent intermediate values. It also invokes the generic `addc` operation from the `generic_machine_ops` module to propagate the carry bit.

Given the abstract definition of multi-word addition, the developer can then generically prove its correctness, as shown with the `big_add_correct` lemma. Notice that the first `ensures` clause says that the result has the expected number of elements, while the second one shows that the addition is computed correctly if each sequence of words is converted into a single big integer value.

As shown in Figure 7, the abstract implementation is a functor parameterized by a machine module. This functor can be instantiated by applying it to a module that refines the formal argument's type. For example, `generic_big_add_impl(bw16_ops)` instantiates a concrete module, which has 16-bit definitions of `big_add` and corresponding 16-bit lemmas such as `big_add_correct`.

3.3 Memory Abstraction

Having written an abstract implementation (§3.2) and instantiated it to specific platforms using functors (§3.1), the Galápagos developer must use the resulting platform-specific proofs to show the correctness of their concrete, hardware-specific implementation. The concrete implementation is ultimately written using the hardware's ISA, formalized in Dafny. As discussed in §2 and shown in Figure 2, the ISA and its formalization operate at a very low-level compared to the abstract implementation and proofs. One particularly challenging aspect is that an ISA typically defines a flat, byte-level memory model. For example, the RISC-V model in Figure 2 maps integer addresses to bytes; this means that a 32-byte write to address, say, `0x400`, affects the four bytes at addresses `0x400`, `0x401`, `0x402`, and `0x403`. Such a model is much harder to reason about than the high-level immutable sequences used in the abstract implementation, since the developer must carefully maintain invariants about which memory regions contain which data, and carefully prove at every memory operation that they are accessing the intended data.

To simplify this reasoning and bring the concrete implementation closer to the abstract implementation, Galápagos generalizes prior one-off memory abstraction techniques [11] by providing automatic support for abstracting an ISA's memory model. Specifically, Galápagos uses a functor to define a generic higher-level interface with a structured heap and stack.

As shown in Figure 8, the abstract heap maps an address to a sequence of `uint` words, whose size is specified by the developer. Similarly, the abstract stack is a sequence of frames, where each frame is also a sequence of words. The abstraction layer soundly preserves invariants showing that operations over structured memory are accurately reflected in the underlying byte-oriented memory.

As with the abstract implementation, the developer instantiates Galápagos' abstraction layer by defining the size of the memory entries they want to reason about. As we illustrate below for RISC-V, this instantiation enables a richer interface for memory instructions.

Accessing Heap Buffers. Many cryptographic implementations iterate over fixed-size buffers, e.g., while reading a plaintext message. Galápagos's memory abstraction provides an iterator interface to support such access patterns. This interface allows the programmer to reason in terms of word-sized (or larger) reads and writes made to immutable sequences of data. As a result, the developer can directly invoke the definitions and lemmas instantiated from the abstract implementation (§3.1), which is conveniently written in terms of sequences of structured data.

The main iterator type is `iter_t`, which abstracts over a structured heap entry. Its invariant, `iter_inv`, guarantees that the iterator is well formed; for example, it ensures that the heap entry exists, that the current index is within the buffer's bounds, that the buffer's view of that region of memory as a sequence of `uint` words is consistent with the heap's state, and that a given address, `addr`, is consistent with the iterator's index.

Once the generic memory layer is instantiated for a hardware platform, Galápagos wraps the iterator interface around low-level memory accesses. Figure 9 shows the Vale procedure `lw_heap` that corresponds to the underlying hardware's load word instruction (`RV_LW`) from Figure 2. In addition to the underlying instruction's

```
// The abstract heap is a collection of disjoint buffers,
// each accessed in the map by its base address
type heap_t = map<nat, seq<uint>>

datatype frame_t = frame(fp: nat, content: seq<uint>)
// The stack is a sequence of frames
datatype stack_t = stack(sp: nat, fs: seq<frame_t>)
// relation between the byte level memory and structured heap/stack
predicate mem_inv(mem: map<int, uint8>, h: heap_t, s: stack_t)
{
  ...
}

datatype iter_t = iter_t(
  base_ptr: nat, // Start of the heap buffer
  index: nat, // Current index
  buff: seq<uint> // Abstract view of the heap buffer
)

predicate iter_inv(iter: iter_t, heap: heap_t, addr: nat)
{
    iter.base_ptr in heap
  ∧ heap[iter.base_ptr] == iter.buff
  ∧ iter.index ≤ |iter.buff|
  ∧ addr == iter.base_ptr + BASE() * index
  ∧ ...
}
```

Figure 8: Generic Memory Interface. Dafny types for the structured heap, stack, and iterators over the heap's buffers, plus an invariant that connects an iterator to the contents of the heap.

```
procedure lw_heap(dst: reg32, src: reg32, offset: imm12,
  inc: bool, iter: iter_t) returns (iter': iter_t)
  {: instruction Ins32(RV_LW(dst, src, offset))}
requires
  iter.index != |iter.buff|; // Not at the end of the buffer
  // heap is a global state variable of type heap_t
  iter_inv(iter, heap, src + offset);
ensures
  dst == heap[iter.base_ptr][iter.index]
      == iter.buff[iter.index];
  inc ⟹iter_inv(iter', heap, src + offset + 4);
  !(inc) ⟹iter_inv(iter', heap, src + offset);
```

Figure 9: RISC-V Load from Structured Heap. The untrusted `lw_heap` Vale procedure offers a friendlier interface that is proven sound against the trusted ISA-level `RV_LW` instruction (from Figure 2). The proof relies on invariants maintained about iterator validity (shown in Figure 8).

three arguments (`dst`, `src`, `offset`), the wrapped version takes two additional arguments, namely `inc` and `iter_t`. As shown in the `ensures` clauses, the `inc` flag controls whether the iterator should be advanced upon return. The caller of `lw_heap` must show that the iterator is safe (i.e., within its buffer's bounds) and well formed (satisfies `iter_inv`). In exchange, the caller learns (from the first `ensures` clause) that the destination's value has been updated to reflect the value in the *structured* heap.

In other words, the caller can reason about the contents of the immutable sequences of `uint` words, without worrying about the underlying bytes in the flat memory model. The `lw_heap` procedure returns an updated iterator that is guaranteed to be well formed. This programming style also means that despite all of the complexities in `iter_inv`, the full definition is irrelevant for callers of `lw_heap`, since `lw_heap` maintains the invariant "for free". Figure 10 shows this in action. The procedure `buff_sum` computes the sum of the contents in the buffer pointed at by `a1`. It does so

```
procedure buff_sum(iter: iter_t) returns (iter': iter_t)
  requires
    iter.index == 0 ∧ |iter.buff| == 10;
    iter_inv(iter, heap, a1);
  modifies t1, t2, a1, a2;
  ensures
    iter'.index == 10 ∧ iter_inv(iter', heap, a1);
    a1 == old(a1) + 40;
    a2 == old(a2) + sum(iter.buff);
{
  iter' := iter;
  addi(t1, a1, 40); // t1 points to the end of the buffer
  while (a1 < t1)
    invariant a2 == old(a2) + sum(iter'.buff[..iter'.index]);
    // Automatically maintained by lw_heap
    invariant iter_inv(iter', heap, a1);
  {
    iter' := lw_heap(t2, a1, 0, true, iter');
    add(a2, a2, t2); // a2 += t2
    addi(a1, a1, 4); // a1 += 4
  }
}
```

Figure 10: Looping Over a Structured Memory Buffer. A Vale procedure illustrating the use of the iterator interface to ergonomically process heap buffers. The `iter_inv` is maintained for free due to Galápagos abstraction layer design. Slightly elided detail: `sum` is a wrapped sum rather than mathematical sum due to overflow.

via pointer manipulation (e.g., incrementing `a1` by four on each loop iteration), but the correctness of these memory operations is maintained by the iterator `iter'`, which `lw_heap` updates.

Galápagos offers a similar interface, `sw_heap`, that wraps RISC-V's store word instruction. Like `lw_heap`, it takes in and returns an iterator, guarding the heap-buffer writes and maintaining the well-formed property of the iterator.

Accessing Stack Variables. Galápagos' memory abstraction layer also provides a structured stack as a generically-proven abstraction over the byte-level memory. The stack is a sequence of frames, each containing several slots for local variables. This makes it simpler for the implementation to prove that variables spilled from registers to the stack retain their value until the next access. Variables in the current frame can be read through the procedure `lw_stack`, which is another wrapper around the load word instruction (`RV_LW`), except the source-address register is hard-coded to be the stack pointer (`SP`). Stack frames can be added and removed using the procedures `push_stack` and `pop_stack`, which are wrappers around subtraction from and addition to the stack pointer.

3.4 Assembly Implementation

With Galápagos, the developer provides, in Vale, a hardware-specific implementation of their cryptographic primitive. They can do this by transcribing the assembly output by a compiler (e.g., when run on C reference code), by handcrafting the Vale assembly to exploit optimization opportunities missed by a generic compiler, or any mix of these strategies.

As they write their implementation, they interact with memory via the high-level, structured memory interfaces provided by Galápagos (§3.3). This makes it straightforward to invoke the definitions and proofs from the hardware-specific instantiation of the abstract implementation (§3.2). For example, because Galápagos' iterators abstract the ISA's byte-level memory into sequences of structured

```
procedure big_add(x_iter: iter_t, y_iter: iter_t, z_iter: iter_t)
  returns (z_iter': iter_t)
  requires
    x_iter.index == y_iter.index == z_iter.index == 0;
    |x_iter.buff| == |y_iter.buff| == |z_iter.buff| == 96;
    iter_inv(x_iter, heap, a1);
    iter_inv(y_iter, heap, a2);
    iter_inv(z_iter, heap, a3);
  modifies
    t1; t2; a1; a2; a3; a4;
  ensures
    iter_inv(z_iter', heap, a3);
    a4 == 0 \/a4 == 1; // Carry out bit
    to_nat(x_iter.buff) + to_nat(y_iter.buff) ==
      to_nat(z_iter'.buff) + a4 * pow(BASE(), 96);
{
  // Implementation code here with loops maintaining iter_inv
  ...
  // Invoke concretized lemma from the abstract impl's proof
  big_add_correct(x_iter.buff, y_iter.buff, z_iter'.buff, a4);
}
```

Figure 11: A Concrete Vale Implementation of Multi-Word Addition. By writing the implementation's pre- and post-conditions in terms of the abstract implementation's definitions (from Figure 7), the developer can easily invoke the corresponding generic lemma concretized to the this platform.

data, the iterators' sequences can be passed directly to the lemmas proven about the abstract implementation.

To illustrate this process, Figure 11 shows an excerpted version of the concrete RISC-V implementation of multi-word addition. It takes in an iterator for each of the x, y, and z buffers. Internally, it uses the `lw_heap` and `sw_heap` procedures to interact with these buffers in terms of the immutable sequences contained in the iterators (e.g., in `x_iter.buff`). This allows the implementation to easily invoke the concretized proof from the abstract implementation (i.e., `big_add_correct` from Figure 7), since both operate over the same high-level sequences. The proof demonstrates that the assembly implementation has successfully computed a step of the abstract implementation (namely computing the sum).

3.5 Algebra Solver Support

Algebraic reasoning is a common theme in cryptographic proofs. The highly parameterized nature of Galápagos also means that many architecture-specific constants cannot be assumed, resulting in formulas with more symbolic components.

Due to the undecidable nature [42] of general non-linear problems, SMT solvers (including Z3, the solver Dafny relies on), while quite effective at many logical theories, often struggle with non-linear reasoning. However, certain sub-classes of non-linear formulas such as congruence relations have been shown to be decidable and robustly handled by dedicated algebra solvers [28].

Inspired by prior work [61] in the interactive theorem prover setting, we have extended Dafny to offer similar support for the Singular algebra solver [18]. A developer can provide a proof goal and relevant facts (proven in standard Dafny) and then explicitly invoke the solver via the new `gbassert` keyword. We provide more details on our encoding in Appendix A.

3.6 Dafny Standard Library Support

Dafny provides a basic set of language features (e.g., sequences or maps) for defining and proving the correctness of an implementation. However, any additional properties must be proven from scratch by the developer. As a result, previous Dafny projects [11, 13, 22, 27, 29, 30, 39, 40] have each developed their own project-specific libraries. This has contributed to significant duplication of effort across projects and even across time, as these project-specific libraries are typically not maintained as Dafny actively evolves.

Early in Galápagos's development, we observed that we would need many of the same properties proven by previous projects, so rather than adding yet another project-specific collection, we have created the first Dafny standard library. The library offers a collection of definitions and lemmas, all fully verified with the latest version of Dafny. They cover data structures (e.g., maps, sequences, and sets), parameterized big integers represented as multi-limb sequences, and an extensive non-linear algebraic properties for dispatching problems algebra solvers (§3.5) cannot handle.

In creating the new library, we drew upon code and proofs from past projects, but rewrote them in a uniform style (both syntactically and in proof style). We also extended them to fill in obvious gaps. The main components covered by our version is discussed below.

Data Structures. Dafny provides built-in support for sequences, maps, and sets, making them convenient for modeling a wide variety of systems. On top of these functional data structures, we added more robust support for performing and reasoning about insertion, removal, extrema, subsequencing or subsetting, conversions between data structures, and higher-order functions (fold, filter, etc.) over the data structures.

Big Integers. As discussed earlier, cryptographic algorithms often operate on large integers that cannot fit into a single machine word. We provide a parameterized library for representing such large integers as multi-limb sequences. The library includes operations such as `big_add` shown in Figure 7, lemmas about results of the operations, and lemmas describing the effect of converting between large integers represented by different bases. The latter simplify the reasoning about, say, converting the representation of a number as a sequence of bits into a sequence of 32-bit words.

Non-linear Arithmetic. As discussed earlier, another common theme in cryptographic proofs is algebraic reasoning. While fragments of non-linear reasoning can be decided (as we do with our newly added Singular support – §3.5), the problem as a whole is undecidable. SMT solvers rely on various heuristics to nonetheless try to solve at least some non-linear problems. Unfortunately, in our experience (and that of previous work [22, 30]), such heuristics are unreliable; they can fail to solve seemingly simple problems, and even when they succeed one time, the proofs can break in response to seemingly minor perturbations, even something as simple as variable renaming. To mitigate these effects, our library proves a set of common algebraic properties from first principles and make them available as lemmas. These lemmas are exposed with varying levels of automation built in. Users can invoke very general lemmas (e.g., exposing lots of properties about multiplication), which provide significant automation but may create proof performance problems. Alternatively, developers can invoke tailored lemmas that specify one property (e.g., multiplication is commutative) or even choose a version where they specify exactly which variables in an equation the property should be applied to (e.g., they can specify x and y as arguments to the lemma to show that $x * y == y * x$). These more specific versions require more manual developer work but they provide consistently provide fast, deterministic performance.

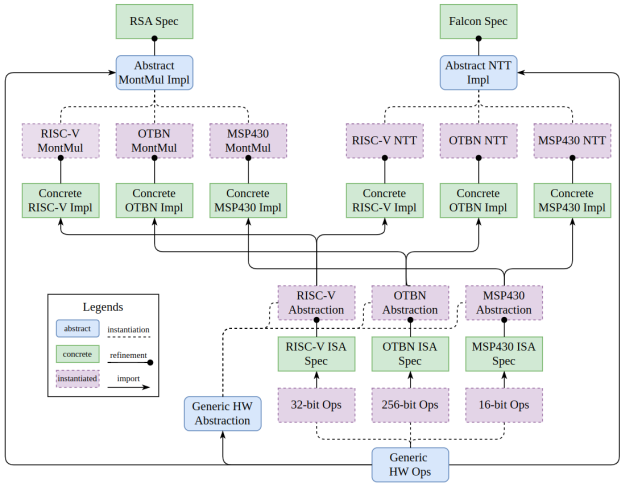

Figure 12: Case Studies Overview. We include three hardware platforms §4.1 and two algorithms §4.2 in our case studies.

The library has been adopted by the Dafny team at Amazon, who have added it to Dafny's continuous integration tests, which run on each commit to the main Dafny repository. The presence of a unified standard library has already encouraged additional contributions from other Dafny developers, including support for monadic operations, searches, sorts, and a Unicode library.

4 CASE STUDIES

As discussed in §1, Galápagos' initial case studies were motivated by the need to support the secure boot of the OpenTitan security chip [49]. OpenTitan aims to process RSA signatures on both the main RISC-V core and on the custom OTBN accelerator. Having both implementations provides a fallback in case the OTBN accelerator is later discovered to have a flaw, or if manufacturers decide to omit the OTBN to save cost and energy. The RSA signature verification routine is used to validate the firmware's integrity at the very beginning of the boot process; this code is burned into the chip's boot ROM, so it cannot be updated through software or microcode patches, only by recalling the chip, designing a new ROM mask, and manufacturing new chips. Hence, the security and correctness of the implementation is crucial.

To further test Galápagos' expressivity, we added yet another hardware platform, the MSP430. We also added a second, lattice-based cryptographic primitive, Falcon, recently standardized by the NIST post-quantum competition.

In this section we elaborate on both the hardware platforms and our verified implementations. Figure 12 illustrates how our case studies exercise the development process from Figure 4.

4.1 Case Studies: Hardware Platforms

Our case studies target three ISAs operating at different bit-widths, using different addressing modes, and supporting different arithmetic operations. We have developed formal semantics for each ISA in Dafny. These semantics are trusted, but we increase our confidence in them by running fuzz tests that compare the output of our semantics with those produced by reference simulators.

MSP430 is a microcontroller family developed at Texas Instruments [10]. It offers a minimalist 16-bit ISA with only 27 instructions (omitting, for example, multiplication). MSP430 memory is byte addressable, and its instructions have six possible addressing modes: register, indexed, absolute, indirect register, indirect auto-increment, and immediate.

RISC-V is an open standard ISA family [57, 63]. For our case study, we use RV32IM, which is the 32-bit base integer ISA (47 instructions) with extensions for integer multiplication and division (8 instructions). The instruction set is quite standard, with a 32-bit address space and byte addressable memory. There are only three data addressing modes: register, immediate, and indexed. One interesting wrinkle is that unlike most platforms (including our other two) RISC-V does not have a dedicated flags register for zero, overflow, or sign bits; instead the developer is expected to check for such conditions using standard ALU operations.

OTBN is a cryptographic accelerator ISA from the OpenTitan project led by lowRISC. OTBN operates on 32 control registers, each 32 bits wide, and 32 data registers, each 256 bits wide. Hence, the data registers alone can potentially hold 1KB of data without any memory accesses. OTBN is designed to accelerate cryptographic computations involving large integers, such as those used in RSA or elliptic curve cryptography. OTBN supports 57 instructions, many of which offer configurable options. For example, the `BN.MULQACC` instruction performs a quarter-word (64 bit) multiplication and then adds the result to a dedicated accumulation register. The instruction can be customized to choose different quarter words from each source/destination register, to shift the multiplication result before accumulating it, and to clear the accumulation register before adding the result.

For the data-memory instructions, `BN.LID` and `BN.SID`, a control register provides the index of the data register as an operand, indirectly reading and writing the wide registers. The instructions read/write 256-bits of data memory and support indirect addressing modes with auto-increment.

Memory Abstractions. Despite the differences in bit-width, memory size and addressing modes, Galápagos' common memory abstraction applies smoothly to all of the hardware platforms.

Instantiating the Galápagos structured memory for each is simple. For each platform, the developer only needs to specify the maximum memory size, the stack size, the word size, and the types for heap entries. Given these definitions, Galápagos automatically generates the high-level memory interface (§3.3), as well as refinement proofs showing that the interface is sound with respect to the byte-level memory model in the trusted ISA semantics. The developer wraps the generated abstractions around platform-specific instructions and uses those to write the platform-specific implementations.

We return to `lw_heap` pattern in Figure 9 for an example in RISC-V. The actual `RV_LW` instruction (from Figure 2) only supports register plus immediate addressing mode. This can be made compatible with the iterator interface by combining `lw_heap` with an explicit `addi` instruction to increment the pointer, or by simply setting `inc` to `false`.

The indirect auto-increment mode in the MSP430 uses a register operand as a pointer, and it increments the pointer after performing the load. This matches the programming pattern that moves the iterator of an array to the next entry after reading the current entry.

There is a similar story on OTBN load instruction. The full syntax of the instruction is:

```
BN.LID <grd>[<grd_inc>], <offset>(<grs>[<grs_inc>])
```

Both `grd` and `grs` are 32-bit control registers, where `grd` specifies the index of the wide register to use as a destination, and `grs` along with the `offset` specifies the source memory address. Suppose that `grd` is register `x1`, which contains the value `0x3`, `grs` is register `x16`, which contains the value `0x8000`. With no offset, this instruction will load the 256-bit word at address `0x8000` into data register `w3`. We note that there are options to increment the control registers, which also correspond to the `lw_heap` iterator pattern.

4.2 Case Studies: Cryptographic Algorithms

RSA. RSA signatures are simple to specify in terms of modular exponentiation of integer values. RSA implementations, however, are amenable to a wide variety of algorithmic and assembly-level optimizations. The algorithmic optimizations are quite complex to reason about even in isolation, let alone in the midst of a complicated assembly-level implementation. Hence Galápagos' split of these obligations between the abstract implementation and the hardware-specific implementation simplifies our correctness proofs.

Abstract Implementation. Our abstract implementation, following the style of OpenTitan's unverified baselines, employs the Montgomery multiplication algorithm [43] to efficiently implement modular exponentiation. Algorithm 1 shows the pseudocode of the algorithm. Notably, the algorithm (and our abstract implementation) is parameterized over both by the radix (e.g., the machine-word's upper limit) and by the size of the big integers, which are represented by sequences of machine words, like the multi-limb sequences in §3.2.

Notice that Line 3 of the algorithm accumulates an intermediate result and requires several multi-limb operations (e.g., $u \cdot m$ is a product between a multi-limb sequence m and a machine word, which produces a multi-limb result, and similarly for $x[i] \cdot y$). Therefore, in the abstract implementation, this line translates into a loop, which handles the element-wise products and sums.

We show our abstract Montgomery multiplication implementation correct by proving the following facts: **(a)** the output is congruent to xyb^{-n}, and **(b)** it is bounded by m. To prove those, we need to construct appropriate loop invariants. For example, in the loop over i starting on Line 1, two invariants are $a \equiv x[..i]yb^{-i} \pmod m$ and $a < 2m$. While the congruence proof above fits perfectly into the subset handled by the extension to Dafny (§3.5), the bound proof does not. Thus for the latter part we rely on lemmas about non-linear arithmetic from our new Dafny standard library (§3.6).

Below we expand on the proof of invariants in the main loop of Algorithm 1, starting from Line 1. The two main invariants are the congruence relation and the bound. i.e. $a \equiv x[..i]yb^{-i} \pmod m$ and $a < 2m$. Consider i^{th} iteration of the loop. We can show that the accumulation preserves the bound:

$$\begin{aligned}
&(a + x[i] \cdot y + u \cdot m)/b \\
\leq\ &(2m - 1 + x[i] \cdot y + u \cdot m)/b \\
\leq\ &(2m - 1 + x[i] \cdot (m-1) + u \cdot m)/b \\
\leq\ &(2m - 1 + (b-1)(m-1) + (b-1)m)/b \\
=\ &(2bm - b - 1)/b \ \ < \ \ 2m
\end{aligned}$$

The congruence proof roughly follows these steps:

$$\begin{aligned}
&(a + x[i] \cdot y + u \cdot m)/b && \pmod m && (1)\\
\equiv\ &(a + x[i] \cdot y + u \cdot m)b^{-1} && \pmod m && (2)\\
\equiv\ &(a + x[i] \cdot y)b^{-1} && \pmod m && (3)\\
\equiv\ &(x[..i]yb^{-i} + x[i] \cdot y)b^{-1} && \pmod m && (4)\\
\equiv\ &(y(x[..i]b^{-i} + x[i]))b^{-1} && \pmod m && (5)\\
\equiv\ &(yx[..i+1]b^{-i})b^{-1} && \pmod m && (6)\\
\equiv\ &yx[..i+1]b^{-(i+1)} && \pmod m && (7)
\end{aligned}$$

We prove that the least significant word of $a+x[i]\cdot y+u\cdot m$ is 0, which justifies (2). We also note that (6) is due to the evaluation rule of multi-limb numbers. These invariants, along with the conditional subtraction at Line 6 of the algorithm, ensure correctness.

Concrete Implementations. We ported the existing, unverified RSA implementations for RISC-V and OTBN into Vale. For the MSP430, we compiled a C version and transcribed the resulting assembly to Vale. For our proofs, we instantiate the abstract implementation's functor with hardware-specific modules that specify an appropriate radix for each platform (e.g., 2^{16} for the MSP430). All three modules specialize RSA's integers to 3072 bits, to match OpenTitan's expectations.

Given the lemmas instantiated from the abstract implementation, proving the correctness of the hardware-specific implementations was relatively straightforward, mostly boiling down to proving various hardware-specific bit-fiddling optimizations. The OTBN implementation was relatively easy, since it could fit all of the RSA integers entirely into registers. Its two sets of flag registers simplified carry propagation, and the built-in accumulator register likewise simplified the multi-word computations. The most significant proof challenge was proving that the implementation correctly used the (very complex) `BN.MULQACC` instruction to compute the multiplication of two 256-bit numbers.

The MSP430 and RISC-V implementations resemble one another. Compared to OTBN, both support a simpler multiplication instruction, while RISC-V was complicated by the lack of a flags register.

Falcon. To validate that Galápagos is applicable to other algorithms, we have used it to produce verified implementations of Falcon [23], a post-quantum signature algorithm recently standardized by NIST. Falcon is based on lattices and its security reduces to the short integer solution problem [1], which differs drastically from RSA.

The spec for Falcon is relatively concise, although still more verbose than RSA, since it depends on definitions of polynomial arithmetic. Simplifying a bit, Falcon verifies a signature s over (hashed) message m, using public key pk, by computing

$$s' \leftarrow m - s \cdot pk \bmod q$$

and checking that the distance between s and s' is small. The signature and the public key are treated as polynomials, so the most computationally intense operation is computing the polynomial multiplication (i.e., $s \cdot pk$).

Abstract Implementation. Naively, a polynomial multiplication takes $O(N^2)$ time, but this can be optimized to $O(N \log N)$ using

Algorithm 1 : Montgomery Multiplication

Require:
b is some radix
n is some length
m, x, y are vectors length n with elements bounded by b
a is a vector length $n+1$ with all 0 elements
$0 \leq x, y < m$
$m' = -m^{-1} \bmod b$
Ensure:
$a = xyb^{-n} \bmod m$

1: **for** $i = 0;\ i < n;\ i = i + 1$ **do**
2: $u = (a[0] + x[i] \cdot y[0])m' \bmod b$
3: $a = (a + x[i] \cdot y + u \cdot m)/b$
4: **end for**
5: **if** $a > m$ **then**
6: $a = a - m$
7: **end if**

Algorithm 2 : Number Theoretic Transform (NTT) with Cooley-Tukey (CT) butterfly

Require:
n is a power of two.
q is a prime such that $q \equiv 1 (\bmod\ 2n)$.
a is a vector in $\mathbb{Z}_q^n$ (standard order).
ψ is a primitive $2n$-th root of unity in $\mathbb{Z}_q$
Ψ_{rev} is a vector in $\mathbb{Z}_q^n$ with powers of ψ (bit-reversed order).
Ensure:
a is the NTT of its initial content (bit-reversed order).

1: $t = n$
2: **for** $m \leftarrow 1;\ m < n;\ m \leftarrow 2 \cdot m$ **do**
3: $t = t/2$
4: **for** $i \leftarrow 0;\ i < m;\ i \leftarrow i + 1$ **do**
5: $s = \Psi_{rev}[m + i]$
6: **for** $j \leftarrow 2i \cdot t;\ j < 2i \cdot t + t\ ;\ j \leftarrow j + 1$ **do**
7: $e = a[j]$
8: $o = a[j + t] \cdot S$
9: $a[j] = (e + o) \mod q$
10: $a[j + t] = (e - o) \mod q$
11: **end for**
12: **end for**
13: **end for**

the number theoretic transform (NTT). In our abstract implementation, we employ the Cooley-Tukey (CT) butterfly algorithm [15] to compute a forward NTT operation (shown in pseudocode in Algorithm 2). Notice that the algorithm, like our abstract implementation, is parameterized over the prime q that defines the field and the size n of the polynomials. Hence, our generic NTT implementation can be instantiated for many other lattice-based algorithms beyond Falcon.

While the pseudocode in Algorithm 2 is relatively succinct, the justifications for why each step computes the right value are surprisingly subtle and are described across multiple research papers [37, 38, 44, 45].

```
CLR R10 ; clear R10
SUBC R10, R10 ; subtract with overflow flag
              ; R10 is either 0x0000 or 0xFFFF
AND 12289, R10 ; R10 is conditionally set to Q
```

Figure 13: MSP430 Set Register on Overflow.

```
add a1, a1, a0 ; sum up a0 and a1
sltu a0, a1, a0 ; if the sum is less than a0, set a0
```

Figure 14: RISC-V Extract Carry Bit.

We provide some brief intuition for the algorithm's correctness and refer the interested reader to [38] for more details. The NTT algorithm works with a sequence of words, where each word represents a polynomial coefficient in the ring $\mathbb{Z}_q$. Hence we can think of a sequence as a polynomial and reason about the effect of evaluating it on a point. If we have sequence $a \in \mathbb{Z}_q^n$ and point $x \in \mathbb{Z}_q$, then the evaluation $a(x)$ can be written as $\sum_{j=0}^{n-1} a[j]x^j$. Let ω be the primitive n-th root of unity in the ring $\mathbb{Z}_q$. The NTT algorithm evaluates the polynomial a at the points $\omega^0, \omega^1..\omega^{n-1}$. More formally, $\mathrm{NTT}(a)[i] = \sum_{j=0}^{n-1} a[j]\omega^{ij}$.

The CT butterfly optimization uses the fact that polynomial evaluation can be split into the evaluation of the terms corresponding to even and odd powers. Let the corresponding coefficients be a_e and a_o, then we can rewrite $a(x)$ as $a_e(x^2) + x \cdot a_o(x^2)$. This reduces the problem to to evaluating the polynomials a_e and a_o on the points $\omega^0, \omega^2..\omega^{2(n-1)}$. Since ω is a primitive n-th root, the list now only contains $\frac{n}{2}$ distinct points. Applying this recursively produces the $O(N \log N)$ running time.

Note, however, that for additional efficiency, Algorithm 2 is an iterative and in-place version of the CT butterfly. The loop over m that starts on Line 2 corresponds to the size of the polynomial, which doubles at each level. The loops over i and j combine the evaluations of the smaller polynomials.

Concrete Implementations. Having dealt with the complex mathematical reasoning in our abstract implementation, our concrete Falcon implementations focus on proving that they faithfully execute the operations dictated by the abstract implementation. Of the three implementations, the OTBN implementation is the simplest, since we were able to implement Falcon's many additions and subtractions modulo q by simply loading q into OTBN's dedicated modulus register and then invoking OTBN's modular addition and subtraction instructions. Implementing these operations on the MSP430 and RISC-V was more complex and involved some non-trivial bit manipulation. For example, on RISC-V the carry bit can be extracted through conditional branches, but Figure 13 is more efficient. Figure 14 shows another example from the MSP430. Without using branches, the code conditionally sets `R10` to `12289` (the modulus q) based on the overflow flag.

5 EVALUATION

We aim to evaluate two key questions.

(1) How much developer effort does Galápagos save?
(2) How do implementations developed with Galápagos perform compared to unverified reference implementations?

	Specification		Abstraction		Total	
	loc	savings	loc	savings	loc	savings
Generic	453	-	1,140	-	1,593	-
MSP430	490	48%	1,091	51%	1,581	50%
RISC-V	685	39%	1,273	47%	1,958	44%
OTBN	1,506	23%	1,843	38%	3,349	32%

Figure 15: Hardware Line Counts & Estimated Effort Saved.

	RSA			Falcon		
	Dafny	Vale	savings	Dafny	Vale	savings
Spec	58	-	-	440	-	-
Generic	963	-	-	5,280	-	-
MSP430	32	1,757	34%	290	2,945	62%
RISC-V	446	1,824	29%	543	2,654	62%
OTBN	339	2,103	28%	163	2,641	65%

Figure 16: Algorithm Line Counts & Estimated Effort Saved.

	Line Count	Definitions	Lemmas
Data Structures	1,219	46	40
Big Integers	914	27	29
Nonlinear Arith.	3,732	7	249

Figure 17: Standard Library Statistics.

5.1 Developer Effort

Below, we estimate how much effort is saved by applying Galápagos's abstractions, rather than writing them from scratch for each platform or algorithm. Hence we report the ratio of the generic part to the sum of generic and platform/algorithm-specific parts.

Case Study Hardware. Figure 15 measures the lines of code developed for our three hardware platforms. The generic row contains the abstract machine model (§3.2) and the memory abstraction layer (§3.3). The other rows show the additional lines of code needed to support each ISA's specification and abstraction. OTBN requires slightly more effort due to the complexities of the ISA's design.

The generic row is a one-time cost when developing the Galápagos framework. For the simpler ISAs, it saves up to half of the code that would have been written if developed without Galápagos.

Case Study Algorithms. Figure 16 presents the lines of code developed for our cryptographic algorithms. The specification and the generic implementation are the per-algorithm one-time cost. We note that the generic implementation for RSA is much shorter than Falcon's, largely due to the Dafny standard library's support for big-integer reasoning. For the concrete implementations, the Vale code embeds the concrete assembly while the Dafny code measures the additional platform-specific lemmas needed. Notice that the generic code reduces the proof burden for RSA by ∼ 30% and for Falcon by more than 60% (RSA has a lower ratio due to its heavy use of our standard library).

In our initial verification efforts, we verified implementations of RSA for the OTBN and RISC-V using traditional monolithic techniques from prior work [11, 24, 55]. Motivated by the duplication across these implementations, we then developed the Galápagos framework and used it to refactor the code. This reduced the developer-written platform-specific code by 28% for OTBN and 29% for RISC-V. We then further leveraged the framework to both specify the MSP430 and add a custom RSA implementation, in approximately one week of developer effort.

With Falcon, we had the Galápagos framework in place, so we initially focused on the abstract proofs related to the NTT, which took 4 developer months. We then derived the platform specific implementations in ∼ 1 developer month.

Standard Library. As discussed in §3.6, we introduced Dafny's first standard library. Our case studies make heavy use of it, with ∼300 calls to standard-library lemmas. Figure 17 summarizes various statistics about the new library. Notice that even though the non-linear portion only includes a handful of definitions (primarily for basic recursive definitions of the various non-linear operations), it provides 249 lemmas proving properties of those definitions.

Singular Support. Our case studies invoke Dafny's new Singular solver 27 times, often for properties that would have been quite painful to prove via manual lemma invocations. As evidence for this, we replaced 15 manual proofs with Singular invocations, eliminating ∼ 525 lines of proof code.

5.2 Performance

Hardware Setup. We execute our verified RISC-V and MSP430 code on two physical development boards and compare the cycle counts of our verified code against their unverified baselines. For RISC-V, we use SiFive's HiFive1 Rev B featuring the Freedom E310 microcontroller. We run the controller at the default 16 MHz. For MSP430, we use a Texas Instrument LaunchPad with the MSP430FR2476 microcontroller configured to run at 8 MHz.

Since OpenTitan chips are still working their way through their first production run, to measure performance of our OTBN implementations, we rely on OpenTitan's cycle-accurate simulator [48].

Baselines. For RSA, prior to our work, the OpenTitan team produced a hand-written assembly implementation for OTBN, and they used a C compiler (configured to optimize for size) to produce code for RISC-V. We similarly use a C compiler for the MSP430. These three implementations serve as unverified RSA baselines.

Falcon has pre-existing C implementation [34] but no optimized assembly for the hardware platforms we target. Hence, we rely on a C compiler to produce unverified baselines for RISC-V and the MSP430. No unverified baselines exist for OTBN, so we wrote our verified implementation from scratch.

Results. Figure 18 shows our performance results for our various verified implementations and their unverified baselines. We find that our verified implementations typically perform within ±2% of their respective baseline implementations. This result is expected, since our verified implementations differ from the baselines only in minor ways which make the code more amenable to verification, e.g., instruction reordering. Our verified Falcon implementation for the MSP430, however, is considerably faster than its compiled baseline. We attribute this result to our hand-tuned register allocation in the verified version.

	MSP430	RISC-V	OTBN
RSA			
Baseline	144,998,445	9,355,922	160,814
Verified	142,870,737	9,454,635	160,664
% Change	-1.47%	+1.05	0%
Falcon			
Baseline	2,810,513	846,946	-
Verified	2,015,556	846,926	256,796
% Change	-28.3%	0%	-

Figure 18: Case Study Performance (Cycle Counts). If % Change is negative, the verified version outperforms the unverified baseline. OTBN does not have an unverified Falcon baseline.

6 RELATED WORK

Barbosa et al. present a recent summary of computer-aided cryptography [8]. Here we focus on more closely related work on formally verified cryptographic implementations. We roughly categorize the work by target (source or assembly language) and by technique.

High-Level Languages. Several lines of work verify or produce cryptographic code in high-level languages. For example, some work [5, 9, 65] uses the Verified Software Toolchain [4] and yields C code, as does work on Fiat Crypto [21] and the HACL* library [53, 68]. Other work [59] uses SAW [19] to produce C and Java code. Still other work [67] relies on extraction to OCaml.

All of this work trusts a compiler (often run in a maximally aggressive optimization mode) to correctly and securely produce machine code suitable for execution. Such trust may be misplaced [21, 62, 64]. Relying on a compiler can also be problematic for emerging hardware platforms, like OTBN, for which compilers do not yet exist. Historically, this approach has also produced code that lags hand-tuned assembly by 2× [21] to 100× [67].

Low-Level Languages. Work in Jasmin [2, 3] verifies implementations written in a domain-specific language and then uses verified compilation to produce an executable. Fiat Crypto [21] also employs verified compilation from high-level elliptic curve descriptions to C-level implementations. Subsequent work suggests a path towards extending their verified pipeline to assembly [51, 52]. While attractive, developing a verified compiler (or even a verified backend) is a significant upfront development effort, and it asks engineers to write proofs about compilation passes, rather than about the code they wish to execute. It may also be difficult to generically match the ingenuity that performance engineers put into their hand-crafted assembly.

In contrast to verified compilation, previous work [11, 24, 55] based on Vale [11] directly verify a wide variety of cryptographic algorithms written in assembly. However, that work primarily focuses on x86-64, with a few implementations for Arm. These implementations and their proofs are standalone efforts, with little code or proof shared between architectures, even for implementations of the same algorithm.

Another line of work [14, 54, 61] targets implementations in an assembly-like domain-specific language (translated from platform-specific assembly via Python). The work's key insight is that often the proof of correctness for the core of a cryptographic routine can be automatically partitioned into proofs about basic mathematical operations and proofs about machine behavior (e.g., proving the absence of overflow), with the former discharged by an algebra solver (Singular [18]) and the latter discharged via an SMT solver (Z3 [17]). This work is complementary to Galápagos, which focuses on providing functor-based platform and algorithm abstractions that can be verifiably reused for multi-platform development. Similarly, their work inspired our integration of Singular into Dafny, but we have found that working in a general verifier like Dafny is critical, since it is unclear how to soundly and automatically break up and efficiently discharge the proof obligations that arise from larger implementations that include memory operations, conditional branches, non-linear equations beyond congruence relations, and arbitrary-length sequences needed to compute, say, RSA.

Extracting Common Algorithmic Features. Many verification projects focus on verifying elliptic curve operations, and several have extracted common algorithmic code (e.g., computing over Montgomery curves), either as libraries [67] or as compiler passes (in Fiat Crypto [21]). This generic code is then instantiated for specific curves that may have different optimal strategies for representing curve points. Galápagos also abstracts over the algorithm, but it differs in using verified functors and focusing on implementations of the same algorithm on different hardware platforms, rather than different algorithms/curves on the same platform.

Prior work has also targeted the number theoretic transform, which is the building block of many post-quantum cryptographic algorithms. Navas et al. use abstract interpretation to show that NTT implementations in C are free of algorithmic overflows [44]. Other work has produced verified NTT implementations through domain-specific languages [31, 58]. These works focus on the techniques to facilitate "push button" verification of individual NTT implementations, while Galápagos focuses on amortizing the verification effort across multiple implementations.

7 CONCLUSION

We have presented the Galápagos framework, which aims to lower the cost of developing high-performance cryptographic implementations across an increasingly heterogeneous hardware landscape. Galápagos uses functors to abstract algorithms and platforms, which can then be automatically instantiated across heterogeneous hardware. Using Galápagos to verify six cryptographic implementations of RSA and Falcon on three wildly varying platforms shows that Galápagos reduces the developer's burden without sacrificing performance. OpenTitan is deploying our verified RSA code at scale. Ultimately, we hope Galápagos helps verified cryptography to boldly go where no (verified) cryptography has gone before.

ACKNOWLEDGMENTS

We thank Jay Bosamiya, Joshua Gancher and the anonymous reviewers for their helpful feedback on the paper. We thank our collaborators at OpenTitan team, especially Felix Miller, Jade Philipoom, Dominic Rizzo, and Rupert Swarbrick for their help with the OpenTitan case studies.

This work was funded in part by National Science Foundation (NSF) Grant No. 1801369 and grants from the Intel Corporation and Rolls-Royce. Sydney Gibson was also funded by the NSF Graduate Research Fellowship Program under Grant No. DGE1745016.

REFERENCES

[1] Miklós Ajtai. 1996. Generating Hard Instances of Lattice Problems. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*.

[2] 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 ACM Conference on Computer and Communications Security (CCS)*.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2020. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[4] Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the European Conference on Programming Languages and Systems (ESOP/ETAPS)*.

[5] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *ACM Trans. Program. Lang. Syst.* (April 2015).

[6] arm. 2022. NEON. https://developer.arm.com/Architectures/Neon.

[7] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. In *Proceedings of the ACM Symposium Principles of Programming Languages(POPL)*.

[8] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[9] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *Proceedings of the USENIX Security Symposium*.

[10] Lutz Bierl. 2000. *MSP430 Family Mixed-signal Microcontroller Application Reports*. Texas Instruments.

[11] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proceedings of the USENIX Security Symposium*.

[12] Jay Bosamiya, Sydney Gibson, Yao Li, Bryan Parno, and Chris Hawblitzel. 2020. Verified Transformations and Hoare Logic: Beautiful Proofs for Ugly Assembly Language. In *Proceedings of the Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*.

[13] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: A verified, concurrent, crash-safe journaling system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[14] 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 ACM Conference on Computer and Communications Security (CCS)*.

[15] James Cooley and John Tukey. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comp.* 19, 90 (1965), 297–301.

[16] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

[17] L. de Moura and N. Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[18] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann. 2022. Singular 4-3-0 — A computer algebra system for polynomial computations. http://www.singular.uni-kl.de.

[19] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. 2016. Constructing Semantic Models of Programs with the Software Analysis Workbench. In *Conference on Verified Software - Theories, Tools, and Experiments (VSTTE)*.

[20] Derek Dreyer. 2005. *Understanding and Evolving the Ml Module System*. Ph. D. Dissertation. Carnegie Mellon University, USA.

[21] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[22] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[23] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. 2017. Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU. https://falcon-sign.info/.

[24] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. 2019. A Verified Efficient Embedding of A Verifiable Assembly Language. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*.

[25] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. 1989. The Digital Distributed System Security Architecture. In *Proceedings of the National Computer Security Conference*.

[26] Shay Gueron. 2012. Intel® Advanced Encryption Standard (AES) New Instructions Set. https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf.

[27] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. 2020. Storage Systems are Distributed Systems (So Verify Them That Way!). In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[28] John Harrison. 2007. Automating Elementary Number-Theoretic Proofs Using Gröbner Bases. In *Proceedings of the Conference on Automated Deduction (CADE)*.

[29] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[30] 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 *Proceedings of the USENIX Symposium on Operating System Design and Implementation (OSDI)*.

[31] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2022. Verified NTT Multiplications for NISTPQC KEM Lattice Finalists: Kyber, SABER, and NTRU. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2022).

[32] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security* (2001).

[33] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[34] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen. [n. d.]. PQClean. https://github.com/PQClean/PQClean Commit febf78a.

[35] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. https://doi.org/10.1109/IPSN.2016.7460664

[36] Royson Lee, Stylianos I Venieris, Lukasz Dudziak, Sourav Bhattacharya, and Nicholas D Lane. 2019. Mobisr: Efficient on-device super-resolution through heterogeneous mobile processors. In *Proceedings of the Conference on Mobile Computing and Networking (MobiCom)*.

[37] Zhe Liu, Hwajeong Seo, Sujoy Sinha Roy, Johann Grossschadl, Howon Kim, and Ingrid Verbauwhede. 2015. Efficient Ring-LWE Encryption on 8-Bit AVR Processors. In *Proceedings of IACR Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg.

[38] Patrick Longa and Michael Naehrig. 2016. Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography. In *Proceedings of the Conference on Cryptology and Network Security (CANS)*.

[39] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Haojun Ma, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. 2022. Armada: Automated Verification of Concurrent Code with Sound Semantic Extensibility. *ACM Transactions on Programming Languages and Systems* 44, 2 (June 2022).

[40] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. 2013. Verifying Security Invariants in ExpressOS. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[41] Akhil Mathur, Nicholas D. Lane, Sourav Bhattacharya, Aidan Boran, Claudio Forlivesi, and Fahim Kawsar. 2017. DeepEye: Resource Efficient Local Execution of Multiple Deep Vision Models Using Wearable Commodity Hardware.

In *Proceedings of the Conference on Mobile Systems, Applications, and Services (MobiSys)*.
[42] Yuri Vladimirovich Matiyasevich. 1993. *Hilbert's Tenth Problem.* The MIT Press.
[43] Peter L. Montgomery. 1985. Modular Multiplication without Trial Division. *Math. Comp.* 44, 170 (1985), 519–521.
[44] Jorge A Navas, Bruno Dutertre, and Ian A Mason. 2020. Verification of an optimized NTT algorithm. In *Proceedings of the IFIP Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*.
[45] Hamid Nejatollahi, Nikil D. Dutt, Sandip Ray, Francesco Regazzoni, Indranil Banerjee, and Rosario Cammarota. 2019. Post-Quantum Lattice-Based Cryptography Implementations. *ACM Computing Surveys (CSUR)* 51 (2019).
[46] OpenSSL. [n. d.]. Vulnerabilities. https://www.openssl.org/news/vulnerabilities.html. Retrieved June, 2023.
[47] OpenSSL Team. 2005. OpenSSL. http://www.openssl.org/.
[48] OpenTitan . [n. d.]. OTBN simulator. https://github.com/lowRISC/opentitan/tree/0be5abcf448de4e6076067820e27fbc77bd93a72/hw/ip/otbn/dv/otbnsim.
[49] OpenTitan. [n. d.]. The OpenTitan Project. https://opentitan.org/.
[50] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. 2011. *Bootstrapping Trust in Modern Computers.* Springer.
[51] 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 ACM Conference on Programming Language Design and Implementation (PLDI)*.
[52] 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 *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR)*.
[53] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. 2020. HACL×N: Verified Generic SIMD Crypto (For All Your Favorite Platforms). In *Proceedings of the ACM Conference on Computer and Communications Security*.
[54] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2018. Verifying Arithmetic Assembly Programs in Cryptographic Primitives. In *Proceedings of the Conference on Concurrency Theory (CONCUR)*.
[55] 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 Wintersteiger, and Santiago Zanella-Beguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[56] Alastair Reid. 2016. Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design*.
[57] RISC-V Foundation. 2017. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2. Editors Andrew Waterman and Krste Asanovic.
[58] Ryo Tokuda and Yukiyoshi Kameyama. 2023. Generating Programs for Polynomial Multiplication with Correctness Assurance. In *Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation*.
[59] Aaron Tomb. 2016. Automated Verification of Real-World Cryptographic Implementations. *IEEE Security Privacy Magazine* 14, 6 (Nov. 2016).
[60] Trusted Computing Group. 2011. Trusted Platform Module Main Specification. Version 1.2, Revision 116.
[61] Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2017. Certified Verification of Algebraic Properties on Low-Level Mathematical Constructs in Cryptographic Programs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.
[62] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. 2012. Undefined behavior: what happened to my code?. In *Proceedings of the Asia-Pacific Workshop on Systems*.
[63] Andrew S. Waterman. 2016. *Design of the RISC-V Instruction Set Architecture.* Ph. D. Dissertation. University of California, Berkeley.
[64] X. Yang, Y. Chen, E. Eide, and J. Regehr. 2011. Finding and Understanding Bugs in C compilers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.
[65] 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 ACM Conference on Computer and Communications Security (CCS)*.
[66] Yi Zhou, Sydney Gibson, Sarah Cai, Menucha Winchell, and Bryan Parno. 2023. Galápagos source code. https://github.com/secure-foundations/veri-titan.
[67] Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. 2016. A Verified Extensible Library of Elliptic Curves. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*.
[68] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

A POLYNOMIAL ENTAILMENT DETAILS

We briefly summarize the mathematical underpinnings of polynomial entailment for solving congruence relations (see prior work [28, 61] for more details), follow by our Dafny-specific encoding.

Background: Polynomial Ring Ideals. Fix a set of variables $x_1, ..., x_n$. Let $R = \mathbb{Z}[x_1, ..., x_n]$ be a polynomial ring over the integers; i.e., the elements of R are polynomials with indeterminate $x_1, ..., x_n$ and integer coefficients. A set $I \subset R$ is an ideal in R iff

$$a + b \in I \quad \forall a, b \in I$$
$$a \times c \in I \quad \forall a, b \in I, c \in R$$

Let $B = \{b_1, .., b_m\} \subseteq R$. B is a generating set of an ideal I, or $I = \text{ideal}(B)$ if

$$I = \left\{ \sum_{i=1}^{m} r_i \times b_i \mid r_1, ..., r_m \in R \right\}$$

The ideal membership problem decides if a polynomial $p \in R$ belongs to an ideal $I = \text{ideal}(B)$. If $p \in I$, then there exists polynomials $r_1, ..., r_m \in R$ such that

$$p = \sum_{i=1}^{m} r_i \times b_i$$

Our Dafny Encoding. We added a new `gbassert` primitive that takes in one goal statement and an arbitrary number of supporting statements as inputs. Each statement must be a modular congruence, meaning that each statement i is the form of $a_i = b_i \pmod{m_i}$, where $a_i,\ b_i,\ m_i$ are integer typed expressions.

We start with a pass over the input statements to collect existing variables. For invocations of uninterpreted functions, we introduce fresh variables. For example, `msb(x)` might be assigned a variable name t_0, and any occurrences of `msb(x)` will be replaced with t_0.

We then use all of the existing and new variables to construct elements of a polynomial ring R over the integers. The goal statement (at index 0) is in the form of $a_0 = b_0 \pmod{m_0}$ and is translated to the polynomial $a_0 - b_0$. Each supporting statement $a_i = b_i \pmod{m_i}$ is translated into the polynomial $a_i - b_i + p_i m_i$, where p_i is a freshly introduced variable. Because the supporting statement is proven (by Dafny), forall a_i, b_i, there exists some p_i such that $a_i - b_i + p_i m_i = 0$.

We form a generating set for an ideal B by collecting all of the polynomials from the supporting statements along with the modulus m_0 from the goal statement:

$$B = \{m_0, a_1 - b_1 + p_1 m_1, \ldots, a_k - b_k + p_k m_k\}$$

We then invoke Singular [18] to decide whether the goal polynomial $a_0 - b_0$ belongs to the ideal generated by B. If so, then we conclude that $a_0 = b_0 \pmod{m_0}$. The intuition is that if $a_0 - b_0 \in \text{ideal}(B)$, then there exists polynomials $r_0, ..., r_m \in R$

$$a_0 - b_0 = m_0 \times r_0 + \sum_{i=1}^{k} r_i \times (a_i - b_i + p_i m_i)$$

The p_i values discussed above ensure that the summation on the right evaluates to zero. Hence membership effectively shows that there exists $r_0 \in R$ such that $a_0 - b_0 = m_0 \times r_0$, proving that our original goal congruence holds (i.e., that $a_0 = b_0 \bmod m_0$).