Formally Verified Cloud-Scale Authorization

Aleks Chakarov, *Amazon*, aleksach@
Jaco Geldenhuys, *Amazon*, jgeldenh@
Matthew Heck, *Amazon*, mcheck@
Michael Hicks, *Amazon*, mwhicks@
Sam Huang, *Amazon*, srhuang@
Georges-Axel Jaloyan, *Amazon*, gjaloyan@
Anjali Joshi, *Amazon*, anjalijs@
K. Rustan M. Leino, *Amazon*, leino@
Mikael Mayer, *Amazon*, mimayere@
Sean McLaughlin, *Amazon*, seanmcl@
Akhilesh Mritunjai, *Amazon*, amritun@
Clement Pit-Claudel, *Amazon*, pitclauc@amazon.ch
Sorawee Porncharoenwase, *Amazon*, soraweep@
Florian Rabe, *Amazon*, florrabe@
Marianna Rapoport, *Amazon*, rapopor@
Giles Reger, *Amazon*, reggiles@
Cody Roux, *Amazon*, codyroux@
Neha Rungta, *Amazon*, rungta@
Robin Salkeld, *Amazon*, salkeldr@
Matthias Schlaipfer, *Amazon*, schlaipf@
Daniel Schoepe, *Amazon*, schoeped@
Johanna Schwartzentruber, *Amazon*, jjsch@
Serdar Tasiran, *Amazon*, tasirans@
Aaron Tomb, *Amazon*, aarotomb@
Emina Torlak, *Amazon*, torlaket@
Jean-Baptiste Tristan, *Amazon*, trjohnb@
Lucas Wagner, *Amazon*, lgwagner@
Michael W. Whalen, *Amazon*, mww@
Remy Willems, *Amazon*, rwillems@
Tongtong Xiang, *Amazon*, ttx@
Tae Joon Byun, *Meta*, taejoon@umn.edu
Joshua Cohen, *Princeton University*, jmc16@cs.princeton.edu
Ruijie Fang, *University of Texas at Austin*, rjf@abstractpredicates.org
Junyoung Jang, *McGill University*, junyoung.jang@mail.mcgill.ca
Jakob Rath, *TU Wien*, jakob.rath@tuwien.ac.at
Hira Taqdees Syeda, *University of Melbourne*, hira.syeda@unimelb.edu.au
Dominik Wagner, *NTU Singapore*, dominik.wagner@cs.ox.ac.uk
Yongwei Yuan, *Purdue University*, yuan311@purdue.edu

***Abstract*—All critical systems must evolve to meet the needs of a growing and diversifying user base. But supporting that evolution is challenging at increasing scale: Maintainers must find a way to ensure that each change does only what is intended, and will not inadvertently change behavior for existing users. This paper presents how we addressed this challenge for the Amazon Web Services (AWS) authorization engine, invoked 1 billion times per second, by using formal verification. Over a period of four years, we built a new authorization engine, one that behaves functionally the same as its predecessor, using the verification-aware programming language Dafny. We can now confidently deploy enhancements and optimizations while maintaining the highest assurance of both correctness and backward compatibility. We deployed the new engine in 2024 without incident and customers immediately enjoyed a threefold performance improvement. The methodology we followed to build this new engine was not an off-the-shelf application of an existing verification tool, and this paper presents several key insights: 1) Rather than prove correct the existing engine, written in Java, we found it more effective to *write a new engine* in Dafny, a language built for *verification from the ground up*, and then compile the result to Java. 2) To ensure performance, debuggability, and to gain trust from stakeholders, we needed to generate readable, *idiomatic* Java code, essentially a transliteration of the source Dafny. 3) To ensure that the specification matches the system's actual behavior, we performed *extensive differential and shadow testing* throughout the development process, ultimately comparing against 10^{15} production samples prior to deployment.**

Our approach demonstrates how formal verification can be effectively applied to evolve critical legacy software at scale.

The email addresses of Amazon-affiliated authors end with amazon.com. The work by non-Amazon-affiliated authors was done while they were at Amazon.

I. Introduction

To control access to their data and resources, AWS customers write policies that express fine-grained permissions. A cloud-hosted *authorization engine* evaluates incoming requests against those policies to either grant or deny access. It must ensure that decisions are both *fast*, to not slow down normal processing too much, and *correct*, by adhering to public documentation and meeting customer expectations. This authorization engine is used to secure access to over 200 fully featured services, invoked over 1 billion times per second.

Today, any change to the authorization engine undergoes a rigorous evaluation of both design and implementation,

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00166

leveraging unit and integration testing and multiple rounds of code review. Changes are also reviewed by a central security team and undergo extensive third-party penetration testing. While effective, in 2019 we decided even more assurance was needed to allow our authorization system to evolve rapidly to meet AWS's increasing scope and scale. Indeed, the challenge of ensuring the engine's speed and correctness had increased over time, causing us to wrestle with the question: How do we confidently deploy enhancements to this security-critical component and know that backward compatibility and performance for existing customer workflows are preserved? This challenge is not unique to our engine—it is a core challenge for any critical system developed over a long period of time in response to a growing and diversifying user-base.

We sought to gain this confidence through the application of *formal methods*, which involve writing a *formal specification* and proving that code correctly implements that specification. While the obvious benefit of formal methods is assured correctness, it also endows two other benefits:

- *Understanding*: A precise formal specification is easier to read and understand than an optimized implementation. Thus, we can be more confident that changes we make are the right ones, since they are expressed as simple changes to the specification.
- *Agility*: Formal proofs make changes easier to deploy. This is because proofs give a strong guarantee that changes we make to the production code always precisely match our specification, i.e., they do what we intend. For example, if we want to optimize the engine, we can prove that the optimized version still meets the existing specification. If we wish to add a new feature, we can prove that the new specification is backward compatible with the old one before proving the new code satisfies it.

Over a period of four years we developed a new, formally verified authorization engine, called *AuthV2*, that is behaviorally equivalent to the original engine, called *AuthV1*.[2] The effort split evenly between i) specification, proofs, and implementation; ii) compilation; and iii) testing and validation. *AuthV2* was successfully deployed to production in early 2024. It computes authorization decisions three times faster than *AuthV1*. In addition, using our new formal methods-driven development process, we have already been able to confidently—and swiftly—make changes to *AuthV2* after it was deployed.

We believe *AuthV2* to be the first successful application of formal verification to build a performant, functionally equivalent replacement for a highly scaled, legacy system. The core contribution of this paper is to show what was required, beyond a straightforward application of an existing verification tool, to achieve this result: we had to develop new solutions, integrate multiple tools and languages, and use them at scale. The paper recounts the four-phase process we followed to successfully launch *AuthV2*: reverse-engineer the formal specification (Section IV); build a high-quality verification-aware implementation and prove it correct (Section V); generate idiomatic code from that implementation (Section VI); and validate the generated code for performance and correctness under deployment conditions (Section VII). There are several takeaways from our experience that were key to its success.

First, we needed to rewrite *AuthV1* to be amenable to formal verification. It is non-trivial to formally verify software written in a general-purpose language that has not been designed with formal verification in mind. *AuthV1* is deployed in Java for a JVM-based environment. It relies on features that make formal verification challenging, such as complex exception handling, use of mutable state, and heap manipulation. Therefore, following the lead of several past projects [49], [60] we elected to write *AuthV2* in a language built to support formal verification from the ground up. Our language of choice was Dafny [20], [48], which leverages SMT solving to automate much of the proof task. While Dafny has been applied to several prior projects (e.g., [3], [35]), we present novel insights from using it in practice and at scale, e.g., explaining how we managed the problem of *proof brittleness* (aka proof *instability* [70], [71]).

Second, to compile the verified Dafny code to Java we developed our own *idiomatic compiler* that generates readable and performant code, rather than use Dafny's existing Java compiler. Doing so made *AuthV2*'s Java code reviewable by the operators of *AuthV1*, who were not Dafny experts, helping build their trust in its launch readiness. Idiomatic compilation also makes *AuthV2* easily debuggable in the context of the larger Java codebase it operates within. And it preserves the optimizations expressed in *AuthV2*'s Dafny source code. Our compiler goes beyond the scope of prior idiomatic compilers for verified code [32], [61] by supporting more sophisticated source-language constructs and environment models, and leveraging annotation-based static checks.

Third, while much focus in the formal methods community is on proof/verification technology, it is no less important to be sure your proofs consider the right *specification*. We used extensive systematic, differential testing to ensure the specification was accurate and complete, and thus that *AuthV2* (now and in the future) exhibits no deviations from behavior established by *AuthV1*. In particular, we wrote the initial *AuthV2* specification to be executable, and then extensively tested it against the legacy *AuthV1* on synthetic and logged data. We also shadow-tested *AuthV2* against *AuthV1* on production authorization data, ultimately testing against 10^{15} production samples prior to launch. Doing so was pivotal to ironing out last, corner-case mismatches.

This paper details the four-phase process we followed to build *AuthV2*, highlighting these takeaways along the way. It concludes with a detailed comparison of related verification efforts (Section VIII), and ideas for further work (Section IX).

II. Background

A. Cloud service authorization

The legacy authorization engine of AWS, known as *AuthV1*, is written in Java and is part of a software development kit

[2] *AuthV1* and *AuthV2* are pseudonyms.

(SDK) used by customer-facing teams to authorize requests. It provides the following API:

```
Answer evaluate(List<Policy> ps, Request r)
```

Here a `Policy` is the Java representation of a list of authorization rules, called statements, written in a purpose-built, declarative language. Each statement is a five-tuple (`Effect`, `Principal`, `Action`, `Resource`, `Condition`). It specifies under which conditions a principal is permitted or forbidden to take an action on a resource, corresponding to effect Allow or Deny, respectively. The condition is optional and provides a way to specify constraints using typed operators.

For example, here is a policy containing a single statement which states a user named `eve` may perform read-/write actions on jpg files inside the `eve` folder managed by `storage-service` whose name starts with `photo-`.

```
[{
  Effect: Allow
  Principal: [ user/eve ]
  Action: [ read, write ]
  Resource: [ storage-service:::eve/* ]
  Condition: [
   StringLike: { filename: "photo-*.jpg" }]
}]
```

The `Request` type represents a request to some service API. It is a 4-tuple (P, A, R, C) that asks: "is principal P allowed to take action A on resource R given context C?", where C is a key-value list with additional request data.

The `evaluate` function evaluates the request against all provided policies and returns the `Answer` as `ExplicitDeny`, `Allow`, or `ImplicitDeny`. In particular, a request is allowed if some policy allows and no policy denies it. In the absence of applicable policies, a request is denied implicitly.

The above behavior is publicly documented in semiformal statements such as "An explicit deny in any policy overrides any allows." We call this property *Deny Trumps Allow* and revisit it in the following subsection.

B. Dafny: A verification-aware programming language

After *AuthV1* had been in use for about a decade, we felt that our current software assurance process was reaching its limit. It was becoming increasingly difficult to convince ourselves to deploy complex changes, including optimizations, that might disrupt our many customers due to a fault that got missed, in either the design or the implementation. We decided the best path forward was to develop a *formally verified* version of *AuthV1*: We would write a formal specification that described *AuthV1*'s behavior, and an implementation that we proved conformed to that spec. When we changed the implementation in the future, we would either prove it against the existing spec (e.g., if the change was an optimization), or we would prove it against an updated spec that in turn we proved was backward compatible with the existing one. With the decision made to formally verify, the question became: How?

Our first thought was verify *AuthV1*'s Java source code directly, using a verifier such as OpenJML [17] or KeY [1]. However, at the time these tools struggled with complex usage of Java features present in *AuthV1*, such as exception handling, mutable state, and heap manipulation. When we looked at prior successful verification efforts (Section VIII has details), we found that most leveraged a programming environment—such as Isabelle/HOL [57], the Rocq prover [18], or F* [31]—built from the ground up to support formal proof. Following their lead, we elected to rewrite *AuthV1* in Dafny [20], an open-source programming language with an integrated static verifier, and then compile the resulting code to Java.

Dafny features Java-like declarations and statements such as variable assignments, loops, classes, and inheritance. Moreover, it supports functional features such as algebraic datatypes, higher-order functions, mathematical sets and maps, and a strict distinction between side-effect-ful and side-effect-free parts of the code. One often leverages the latter when writing logical specifications, and the former when writing efficient imperative code, but Dafny allows freely mixing both styles. Dafny specifications are akin to the code contracts in Eiffel [54] consisting of method pre- and post-conditions affixed to methods and loop invariants, and are statically checked by the Dafny verifier.

For example, we can use the following code snippet to specify the `Evaluate` function:

```
function Evaluate(ps: seq<Policy>, r: Request):
    Answer {
  if MatchesDeny(ps, r) then ExplicitDeny
  else if MatchesAllow(ps, r) then Allow
  else ImplicitDeny
}
```

and attach it as a post-condition to the imperative implementation of the function.

Users can write explicit lemmas whose pre-/post-condition pair represents an implication about the inputs. For example, we can formalize the deny-trumps-allow property as:

```
lemma DenyTrumpsAllow(ps: seq<Policy>, r: Request,
    p: Policy, s: Statement)
  requires p in ps && s in p.statements &&
    Matches(s, r) && s.effect == Deny
  ensures Evaluate(ps, r) == ExplicitDeny
{ /* proof */ }
```

Dafny's verifier leverages external SMT solvers to discharge proof obligations, as well as manual proofs when automation reaches its limits. Dafny can compile the behaviorally verified code into various target languages. It also provides a foreign-function interface that makes it possible to interface with pre-existing target-language code. While its ecosystem is not nearly as rich as, e.g., Java's, it includes advanced tools like a test generator and a specification auditor.

III. Overview

How does one evolve a legacy system to use formal methods? Our approach took place in roughly four phases: 1) developing a formal specification; 2) building a performant, proved-correct implementation; 3) compiling the implementation to idiomatic, deployable code in the programming language of the legacy system, and 4) deeply validating that implementation under deployment conditions. As discussed in

Section II-B, our particular experience with *AuthV2* leverages Dafny: a Dafny implementation $AuthV2^{\text{impl}}_{\text{dafny}}$ is proved correct against a specification, $AuthV2^{\text{spec}}_{\text{dafny}}$, and is translated to $AuthV2_{\text{java}}$, the Java code which runs in production. Subsequent development on *AuthV2* happens at the Dafny level: Changes to the Dafny implementation are re-proved against the specification (which itself can change, subject to proofs of key properties) and then compiled to Java and deployed. We give an overview of each phase we followed to build *AuthV2* here, and discuss each phase in depth in the subsequent sections.

A. Formalizing the specification

The first phase was to develop $AuthV2^{\text{spec}}_{\text{dafny}}$, the formal specification for *AuthV2* that captures *AuthV1*'s behavior. We wanted *AuthV2*'s specification to satisfy three properties. It should be *simple*, so that it is easy to understand; *formal* so that we can prove properties about it; and *executable* so that we can already test the specification against the legacy authorization engine. These properties strengthen our confidence that the specification truly reflects *AuthV1*'s behavior.

To write the specification $AuthV2^{\text{spec}}_{\text{dafny}}$, we inspected *AuthV1*'s Java implementation and documentation (internal as well as public), and expressed what we found using Dafny. We ultimately proved 52 lemmas about the specification that we expected should hold, including the *Deny Trumps Allow* property stated in Section II-A. Failed proofs would have signaled an incorrect specification or potentially unexpected *AuthV1* behavior. We extensively tested *AuthV2* against *AuthV1* using the latter's existing unit tests, and differential tests at the whole-system level, driven by a test generator. Section IV presents our process in detail.

B. Implementing a proved-correct authorization engine

The next phase was to build a performant, verification-aware Dafny implementation and prove that it conforms to the specification. While the specification is purely functional and inefficient, the Dafny implementation can be imperative and aggressively optimized.

Because *AuthV1* runs in a Java environment, *AuthV2* needed to be written against a model of the standard Java libraries. We therefore developed formal Dafny interfaces for Java arrays, lists, strings, etc. and relevant library methods. Proofs of correctness leverage these interfaces, and the final engine links against the real libraries.

We faced two key challenges during this phase. First, we needed to write these library interfaces in a way that bridges the semantics of the purely functional built-in Dafny types for sequences etc. with Java's imperative ones, while also ensuring the Dafny specification of these foreign functions (which amount to axioms from the perspective of Dafny) correctly model the behavior of the Java functions. Second, we needed to impose some engineering discipline to cope with *proof brittleness*, a phenomenon common in automated reasoning-based systems [7], [12], [63] in which apparently inconsequential changes cause previously-working proofs to fail. We discuss these challenges and our solutions in Section V.

C. Generating deployable code

With a proved-correct Dafny implementation in hand, the next phase was to make sure that the implementation's code is deployable. While *AuthV2* is written in Dafny, *AuthV1* is part of a larger system written in Java. Therefore, the ultimate deliverable of the project is Java code, $AuthV2_{\text{java}}$, generated from *AuthV2*. A critical requirement was for $AuthV2_{\text{java}}$ to meet all operational and performance requirements that had been previously gathered for *AuthV1* throughout the usual software engineering process: $AuthV2_{\text{java}}$ has to be human-readable. This ensures that code review can independently ascertain the correctness and security of $AuthV2_{\text{java}}$.

Therefore, we decided early on not to use the existing Dafny-to-Java compiler, which covers all of Dafny but produces Java code whose performance and idiomaticity are hard to predict for the Dafny programmer. Instead, we developed a custom Dafny compiler (see Section VI) for a fragment of Dafny that corresponds to idiomatic Java. It generates $AuthV2_{\text{java}}$ as *readable* Java code, basically as a transliteration of *AuthV2*. Thus, the *AuthV2* developers could anticipate the exact Java code that would be produced, allowing them to optimize for idiomaticity and efficiency. Ultimately, this proved *crucial* to building the trust necessary to launch *AuthV2*.

Table 1 illustrates the size of our Dafny formalization and the generated Java code for *AuthV2* that now runs in production. The ratio of the size of the Dafny specification and proofs to the size of the Dafny implementation is 5 : 1 (cf. 5 : 1 for Ironclad Apps [36] and 20 : 1 for the initial version of seL4 [43]). Additionally, the generated Java code is larger than the compilation-targeted (non-specification) Dafny code because it includes auto-generated methods like `equals` and `hashCode` in Java.

Table 1: *AuthV2* formalization and generated-code size

	Feature	LOC
Dafny	Specification & proofs	11,570
	Library axiomatization	1,557
	Implementation	2,751
	Total	15,878
Java	Generated implementation	3,303

D. Extensive validation under deployment conditions

The final phase was to subject $AuthV2_{\text{java}}$ to real-world conditions prior to launch. To do so, we deployed our authorization engine as a *shadow* in production environments, and differentially tested $AuthV2_{\text{java}}$ against *AuthV1* on production data, comparing their functionality and performance. While our work prior to this phase gave us significant confidence in *AuthV2*, this final phase assured us of a seamless deployment. We shadow-tested throughout 2023, first comparing the two engines on only 0.1% of samples, and steadily raising the sampling rate to 100% as mismatches became less frequent. Doing so revealed corner-case correctness problems where

AuthV2's specification had not modeled all the subtleties of *AuthV1*. We deployed 8 iterations of $AuthV2_{\text{java}}$ to shadow mode and the final iteration agreed with *AuthV1* on 10^{15} production authorization samples, indicating it was ready to launch in production. Moreover, while in-lab differential testing suggested early-on that $AuthV2_{\text{java}}$ outperformed *AuthV1*, profiling on real-world traffic identified additional performance optimization opportunities. Section VII details our validation efforts.

IV. Formal Specification

The first phase of developing a proved-correct version of a legacy system is developing its *specification*, i.e., a formal description of the intended behavior. An often-overlooked challenge is to ensure that the specification is indeed the intended one. We build confidence about the specification's correctness by taking two steps. First, we make it *executable* and validate it using the legacy system's data and tests. Second, we *prove properties* about it that correspond to key statements in the documentation, e.g., those that help the legacy system's users gain intuition about the semantics of the policy language.

A. Writing a testable specification

If we were starting the project today, we would write a purely functional, executable specification directly in Dafny. However, back in early 2020, Dafny lacked a critical feature: a compiler to Java, the deployment language required for *AuthV2*. Java code was essential to integrate the specification with the existing *AuthV1* authorization engine's test frameworks and test generators. To address this limitation, we started two parallel efforts:

- Developing a temporary, purely functional implementation in Scala (chosen for its ability to interface with Java via the JVM) that closely mirrored the Dafny specification. The development took about two months and comprised 1.1 kLOC.
- Building a Dafny-to-Java compiler to automate the generation of Java code from Dafny specifications.

For example, consider the `StringLike` operator [66] implementation in Scala, which enables policies to match request attributes against strings with wildcards:

```
def wildcardMatch(pattern: String, text: String):
    Boolean {
  val escapedRegex = escapeRegexChars(pattern)
  s"^${escapedRegex}$$".r
    .findFirstIn(text).nonEmpty
}
```

To gain confidence in the correctness of the specification, we tested the Scala model in two ways: First, we applied *AuthV1*'s existing unit and integration tests. Second, we used junit-quickcheck [15], [58] to differentially test the Scala model against *AuthV1*.

The latter required developing a *fuzzer* to systematically inject arbitrary input data into the two models. The main difficulty here is randomly generating allowed policy-request pairs: because the request must match the various policy conditions to be allowed, random input data would be denied immediately and exercise only a fraction of the source code. To maximize code coverage, we hand-wrote input generators so that policies and requests match with high probability. Our generators also ensured data satisfy semantic constraints, e.g., that IP addresses contain realistic values, and that service names are realistic.

After about 3 person-months we achieved about 83% instruction coverage and 72% branch coverage, and differential testing uncovered 13 differences between the early version of the specification and *AuthV1* that we were able to correct. This proved extremely valuable because it was not until later in the development cycle that verification and shadow-testing (see below) produced actionable feedback for repairing the specification.

Once the Dafny compiler and FFI models were ready, we ported the specification to Dafny, and repeated validation testing in the same way. Note that the example specification above still used external library functionality for regular expressions, similar to *AuthV1*'s implementation. That was sufficient for testing the specification. After porting to Dafny, we refined the model further to enable verifying it in depth. This involved, e.g., also modeling the exact semantics of string matching without external dependencies:

```
predicate WildcardMatch(pat: string, txt: string) {
  if pat == [] && txt == [] then true
  else if pat == [] then false
  else if txt == [] then pat[0] == '*' &&
    WildcardMatch(pat[1..], txt)
  else if pat[0] == '*' then
    WildcardMatch(pat[1..], txt) ||
    WildcardMatch(pat, txt[1..])
  else (pat[0] == txt[0] || pat[0] == '?') &&
    WildcardMatch(pat[1..], txt[1..])
}
```

B. Proving properties about the specification

We used Dafny to prove 52 lemmas about our specification, such as the *Deny Trumps Allow* lemma mentioned in Section II, properties about expected behavior of string matching (such as case sensitivity), and the semantics of composed conditions. As an example, consider the property that consecutive stars in a pattern can be collapsed into a single star without affecting the outcome of the match:

```
lemma CollapseStar(pat: string, txt: string)
  requires 2 <= |pat|
  requires pat[0] == '*' && pat[1] == '*'
  ensures WildcardMatch(pat, txt) ==
    WildcardMatch(pat[1..], txt)
{ /* automatically discharged proof */ }
```

V. Verified Implementation

After developing the specification and gaining confidence in its correctness, the next phase is to develop a production-ready implementation, and prove it correctly implements the specification. By "production ready," we mean: 1) the implementation enjoys good performance, and 2) the code is available in a mainstream language in idiomatic style, which

makes it easier for operators to analyze, e.g., when debugging an operational issue. We achieve production readiness by writing Dafny code in an imperative style and compiling it to idiomatic Java. The code is written against a hand-crafted environment that maps constructs (types, methods, etc.) in the Java standard library to corresponding Dafny constructs, so the generated code links against those standard libraries. In this section, we discuss the basic approach, how we modeled the Java environment, and challenges we faced when carrying out the proofs.

A. Basic approach

We developed in Dafny an efficient, imperative, verification-aware implementation of the program, and we proved it equivalent to the functional specification. The Dafny implementation has essentially one imperative method for every function in the specification, each with a post-condition that it returns the same value as the function on all inputs. This makes the verification modular, i.e., every implemented function is verified individually.

Each method's body contains the operational program logic as well as *ghost code* [29]. The latter includes loop invariants and assertions that help the prover along in establishing the post-condition. The ghost code is erased during compilation and does not affect execution.

While the *specification* uses Dafny's built-in types exclusively, the Dafny *implementation* also makes use of models of external types. We define each modeled type with a function `ToSpec()` (Listing 2) which lifts the type to a built-in Dafny type. Methods from external libraries (e.g., the Java standard library) are stubbed out as body-less methods with pre-/post-conditions that model the functionality of the external methods. We explain how we model these external dependencies in Section V-B.

For instance, consider the specification of `WildcardMatch` defined in Section IV-A. It has exponential worst-case complexity whereas our imperative Dafny implementation (excerpt in Listing 1) is only quadratic. The complete implementation

Listing 1: Verified imperative Dafny implementation of $O(n^2)$ wildcard matching algorithm

```
method wildcardMatch(pat: String, txt: String)
  returns (res: bool)
  ensures res == WildcardMatch(pat.ToSpec(),
    txt.ToSpec()) {
  if pat.length() == 0 { return txt.length() == 0; }
  var i: nat31, j: nat31 := 0, 0;
  while i < txt.length()
    /* loop invariants */ {
    if j < pat.length() && pat.charAt(j) == '*' {
      j := j + 1;
  ...
}
```

of this method requires 220 lines of Dafny, 111 lines of which is ghost code. The example takes inputs of type **`String`**, which models the Java standard library string class and its methods, and is different from Dafny's built-in **`string`**, which is a type synonym for a Dafny sequence of characters. By calling `ToSpec()` on **`String`**, we lift it to the corresponding specification type, a Dafny **`string`**, and prove equivalence of the results.

B. Interfacing with the Java target language

Every large software project needs to interface with external dependencies. The standard Dafny compiler compiles built-in types to types defined in a Java runtime library (DJLIB) which comes with the Dafny distribution. One then writes wrapper methods in Java that translate DJLIB types to native Java types used in APIs of the external dependency. The types in DJLIB preserve the properties (such as immutability) and APIs of the Dafny types, which makes compilation simple. For instance, a Dafny **`seq`** is compiled to **`DafnySequence`**, and one would manually translate the **`DafnySequence`** to, say, a **`LinkedList`** expected by an external API. However, Dafny's generated code can be slow compared to handwritten, optimized Java code that uses native Java types, and the wrappers that translate DJLIB types to native types add additional overhead.

We initially considered compiling built-in Dafny types to native, efficient Java types—for instance, compiling **`seq`** to a **`LinkedList`**. This would make the compiler implementation difficult—the compiler needs to make sure that the translation from pure Dafny to mutable Java collections is sound, while also generating idiomatic code. To ensure soundness while preserving idiomaticity, operations on Dafny collections needed to be restricted, which, in turn, limited expressivity. In addition, compiling native Dafny collections did not allow the Dafny programmer to choose different concrete collection implementations in Java: for example, Dafny has a single pure sequence type, but in Java one might choose an **`ArrayList`** instead of a **`LinkedList`** based on the specific requirements of an algorithm.

We opted to address the above problems by defining bindings for Java types and axiomatically modeling their behavior in Dafny. Our compiler (discussed in more detail in Section VI) disallows use of Dafny's built-in types in Dafny implementation code (not in the erased ghost code), unless there is a direct counterpart in Java (e.g., **`bool`**). For illustration, Listing 2 shows an excerpt of how we model **`ArrayList`** and its supertype **`List`** in Dafny, which we use instead of **`seq`** in Dafny implementation code. The `base` field, whose type is the lifted Dafny type, models the current value of the mutable Java object. It is used in the pre-/post-conditions of the modeled methods. We changed Dafny's default `{:extern}` behavior so that Dafny identifier paths can be rewritten to Java ones from the Dafny source code, without needing any handwritten Java code to interface with external dependencies.

Working with these types is ergonomic, it is easy to write the Dafny code in a way that produces the Java code the developer expects (see Fig. 1). Adding new extern Dafny types as needed is easy, and does not require changes to the compiler (see Section VI for exceptions). Additionally, the generated

Listing 2: Modeling Java `List` and `ArrayList` in Dafny. `nat31` is a bounded subtype of Dafny's `int` and compiles to Java's `int`.

```
module {:extern "java.util"} Util {
  trait List<T> extends object {
    ghost var base: seq<T>
    function {:axiom} size(): (res: nat31)
      requires |base| <= nat31_MAX
      reads this
      ensures res == |base|
    ghost function ToSpec(): seq<T> reads this {
      base }
  }
  class ArrayList<T> extends List<T> {
    constructor {:axiom} ()
      ensures base == []
    method {:axiom} addAll(x: List<T>)
      modifies this
      ensures base == old(base + x.base)
  }
}
```

Figure 1: A small Dafny implementation method on the left with generated Java code on the right shows that closely modeling the target language leads to predictable generated code.

```
method cloneToAL(l:
    List<nat31>)
  returns (r:
    ArrayList<nat31>)
  ensures l.ToSpec() ==
    r.ToSpec()
{
  r := new
    ArrayList<nat31>();
  r.addAll(l);
}
```

```
ArrayList<Integer>
    cloneToAL(List<Integer>
    l)
{
  ArrayList<Integer> r =
    new ArrayList<>();
  r.addAll(l);
  return r;
}
```

Java code can directly interface with extern dependencies. No wrappers for translating the generated Java types are needed, and no runtime cost is incurred. This keeps the compiler simple, the Dafny code ergonomic, and the compiled code readable and efficient. In total, we wrote around 1,500 LOC modeling dependencies with their types and APIs as needed for *AuthV2*.

C. Challenge: Proof brittleness and continuous integration

As the project grew, *proof brittleness* became a significant challenge, echoing a well-identified burden in other large-scale verification efforts [7], [12], [63]. This brittleness can be informally described as proofs breaking due to unrelated changes in the environment, tool versions, or code base. The consequences were severe: it slowed the team down, added overhead in fixing broken proofs, and in some cases even prevented the merging of code updates until proofs were repaired. Ultimately, most code or environment changes broke at least one proof, causing delays in deployment by up to 14 days. We implemented three levels of mitigation against proof brittleness.

First, by using a centralized cloud-based infrastructure, we could ensure a standardized and consistent environment across all verification tasks, eliminating environment-based proof variability. Furthermore, by leveraging the massively parallel capabilities of the infrastructure, we could keep the wall-clock verification time from 30 to below 10 minutes, significantly speeding up code reviews and deployments.

Second, we added brittleness checks to our continuous integration, continuous deployment (CI/CD) pipeline to enforce proof stability across toolchain updates and code changes. Those checks raise a warning if a given brittleness threshold is exceeded. The Dafny documentation gives a measurement of brittleness for a VC as the coefficient of variation of resource counts—an abstract measurement of verification cost that depends only on the input formula and exact SMT solver configuration [20]—for different seeds passed to the solver. At its maximum, 41 verification conditions (VCs) were flagged as brittle. Additionally, we designed and implemented a new proof-oriented report generator, detailing the brittleness levels of all VCs. This report is reviewed during pull-requests, and prevents merging commits that would significantly degrade the stability of proofs.

Third, we refactored the most brittle proofs to significantly reduce their resource count, using a set of Dafny guidelines for reducing dependence on automation [53]. An interesting find is that reducing resource count of a proof is a good strategy for reducing its brittleness. Those guidelines consist mostly of making functions `opaque`, which hides their definitions, and using the `reveal` statement, which manually re-introduces them to the verification context.

As an experiment, we gathered metrics on line-of-code (LOC) count, and the ratio of proof versus Dafny implementation for our string processing module, before and after refactoring. In Dafny before refactoring, the module was 800 lines long, of which 300 were dedicated to proofs. After refactoring, the module increased to 1200 lines, of which 1100 were proofs. Overall, this reduced the resource count by two orders of magnitude on the brittle parts of the code base. This led to a sharp reduction of brittle VCs in our project, from 41 to 4.

VI. Idiomatic Compilation

Dafny's standard compiler was designed to favor a general compilation process that can support multiple target languages, including C#, C++, and Java. Unfortunately, this design means that it tends to produce Java code that is neither *readable*, *performant*, nor *maintainable*, which are requirements for *AuthV2*. Having readable Java code allows experts in authorization to review the logic without needing to understand Dafny. Ensuring the Java is performant is a baseline expectation: We wanted *AuthV2* to match or exceed *AuthV1*'s performance. Maintainability has two aspects. First, the generated code must be compatible with various versions of Java and Java libraries, and support standard error-handling mechanisms, such as exception handling used in *AuthV1*. This keeps the experience for end users consistent with *AuthV2*. Second, traceability between the Java and Dafny implementations is important during debugging, especially when dealing with emergent issues that involve complex interactions between client code and *AuthV2*.

To meet our readability, performance, and maintainability requirements, we developed a separate Dafny-to-Java compiler that is able to produce idiomatic output, i.e., output that looks human written. In essence, the compiler works by transliterating the Dafny source into Java, thus preserving Java-style idioms expressed in the Dafny, without introducing temporary variable names or corrupting control-flow structures. As a result, the Dafny developer is able to ensure that the final Java code is of good quality.

A. Non-idiomatic code

Figure 2 shows a snippet of the Java output generated with the standard Dafny compiler [20]. The generated code is hard to review or debug due to its use of boxed types, extra variables, opaque generated names, redundant parentheses, and excessive public access modifiers. In addition, Dafny-generated code can be inefficient because Dafny's abstractions do not always naturally map to preferred abstractions in Java. For example, Dafny types such as `Option` and `Result`, commonly used in functional programming, are compiled to wrapper classes. The extra pointer indirection adds overhead. Moreover, Dafny programs must pay the cost to translate from exceptions to `Result` when interfacing with external Java libraries with exception-throwing APIs. As another example, Dafny uses purely-functional collection implementations rather than imperative ones, such as those found in Java's standard libraries. As a result, the compiler does not leverage Java's (optimized) collection classes.

B. An idiomatically compilable fragment of Dafny

To address these issues, we wrote an idiomatic compiler whose design follows three key principles:

- *Restrict support to subset of source language*: Allow only a subset of Dafny as input, which we call *DafnyLite*. DafnyLite drops support for idiosyncratic Dafny features and retains features that are straightforward to embed into Java. This reduces the complexity of the compilation compared to Dafny's built-in compiler, and thus increases DafnyLite's compilation trustworthiness.
- *Perform idiomaticity transformations*: Apply small, single-purpose transformations that make the output more idiomatic.
- *Model target language in source language*: Model Java-specific features and libraries in Dafny and develop special transformations that pick up on these models. For example, code manipulating a Dafny `Exception` trait is compiled into Java exceptions.

Thus, by defining a subset of Dafny that closely models the intended Java code, the DafnyLite compiler achieves readability and performance, while avoiding complex optimizations.

The DafnyLite compiler is written in F#. It links against the Dafny compiler written in C#, which handles parsing, type checking, name resolution, and desugaring [20]. The DafnyLite compiler then converts the Dafny abstract syntax tree (AST) to the *DafnyLite intermediate representation* (*DLIR*) which includes only expression fragments relevant to Java compilation and excludes ghost code. The right side of Figure 2 compares this compiler's output to the standard one.

The compiler performs a series of 30 self-contained, type-preserving *DLIR*-to-*DLIR* passes. Each pass executes a fine-grained action such as transforming a Dafny feature to a Java feature, simplifying Java code, or increasing the idiomaticity. For example, the compiler consolidates variables, adjusts visibility annotations, shortens identifier paths, flattens nested blocks, and inverts if-conditions when it reduces visual distraction. The resulting *DLIR* syntax tree contains only Java idioms and is printed out as Java source code that is formatted and compiled with `javac`.

The *DLIR* transformations also serve to enforce the restricted DafnyLite subset. For instance, DafnyLite supports only a restricted subset of algebraic datatypes that map easily to Java classes or enums. It limits `Result`-typed expressions to return positions or immediate pattern matching, ensuring sound and idiomatic compilation to Java's exceptions and try-catch constructs. To avoid Java's performance overhead from anonymous functions, it does not support Dafny lambdas. Instead of compiling Dafny's immutable collections, we explicitly model Java library collections in Dafny, ensuring interoperability, readability, and performance. Additionally, the compiler avoids syntactic-sugar constructs, like assignments with object destructuring, that can be easily encoded differently, to reduce complexity.

To ensure idiomatic output, the DafnyLite compiler performs transformations on *DLIR* to adhere to Java conventions. Even with direct Java analogues, some Dafny features need specific transformations to be idiomatic. For example, Dafny datatypes with only nullary constructors become Java enums, `Result` expressions turn into try-catch blocks, and `Option` types, under restrictions, compile to nullable types.

C. Translation validation and fuzzing

Producing idiomatic target code is more complex than simply generating correct executable code. To add confidence that we are preserving Dafny's correctness guarantees, we invest extra effort into validation and testing beyond unit and regression tests.

We use the Java Checker Framework [13] to validate correctness properties that should be ensured by the translation. The Checker Framework enhances Java's type system through annotations; we use it to confirm the 1) absence of null-pointer exceptions, 2) absence of unintended side effects [40], 3) correct usage of signed computations [51], and 4) correct usage of reference equality [25].

Additionally, we employ fuzzing and differential testing, inspired by Irfan et al. [41], to validate that the DafnyLite compiler follows the semantics of the standard Dafny compiler. We use XDSmith [41] to randomly generate and compile programs, ensuring their compilations are correct. The non-idiomatic compilers from the open-source distribution of Dafny serve as the ground truth in our differential testing setup. This approach helped us identify and fix multiple bugs in the DafnyLite compiler.

```
public static boolean wildcardMatch(
  dafny.DafnySequence<? extends dafny.CodePoint> pattern,
  dafny.DafnySequence<? extends dafny.CodePoint> text) {
  boolean r = false;
  if ((java.math.BigInteger.valueOf((pattern).length())).signum() == 0) {
    r = (java.math.BigInteger.valueOf((text).length())).signum() == 0;
    return r;
  }
  int _0_i; int _1_j;
  int _rhs0 = 0; int _rhs1 = 0;
  _0_i = _rhs0; _1_j = _rhs1;
  while ((_0_i) < ((text).cardinalityInt())) {
    if (((_1_j) < ((pattern).cardinalityInt())) &&
        ((((dafny.CodePoint)((pattern).select(_1_j))).value()) == ('*'))) {
      _1_j = (int) ((_1_j) + (1));
      ...
```

```
static boolean wildcardMatch(
  String pattern,
  String text) {

  if (pattern.length() == 0) {

    return text.length() == 0;
  }
  int i = 0;
  int j = 0;

  while (i < text.length()) {
    if (j < pattern.length()
      && pattern.charAt(j) == '*') {
      j++;
      ...
```

Figure 2: Excerpt from standard-Dafny compilation of Listing 1 (left) and the corresponding DafnyLite-generated code (right)

VII. Validating for Deployment

The third and final phase is to validate that *AuthV2*$_{\text{java}}$ matches *AuthV1*'s behavior under deployment conditions, using extensive production data. Doing so has several benefits. First, it validates that the specification truly is accurate and complete. Second, it validates that the Dafny verifier's proofs are sound. Third, it validates that the semantics of *AuthV2* were preserved by our idiomatic compilers when generating *AuthV2*$_{\text{java}}$. Every mismatch between *AuthV2* and *AuthV1* can be investigated to determine the root cause and response, e.g., a change to the specification, a bugfix to a tool, etc. Fourth, the testing reveals performance under production workloads, and helped us identify opportunities for further *AuthV2* optimizations. Finally, it helped earn trust with stakeholders that not only was the deployed system correct, but that it also integrated with the build and production systems of the 200+ services that use it.

We used *shadow testing* [55], [64], [65], a technique where a candidate system runs alongside the existing system in production, to collect the data necessary to convince ourselves *AuthV2*$_{\text{java}}$ was ready for production.

A. Shadow testing

Earlier phases of development used traditional testing and synthetic fuzzing inputs to compare *AuthV1* and *AuthV2*$_{\text{java}}$. Although these approaches were effective at finding issues, our concern was that we were not seeing rare, but meaningful, differences that would impact customers at a scale of 1 billions requests per second. Shadow testing addressed this shortcoming by comparing *AuthV2*$_{\text{java}}$ to *AuthV1* on production data as it was occurring, trillions of times per day, and emitting metrics if a difference was encountered.

In total, shadow testing exposed 7 mismatches owing to inaccuracies or incompleteness of our specification; no issues were found with tools or proofs. As an example, one mismatch exposed a difference in how *AuthV1* and *AuthV2*$_{\text{java}}$ handled Date and Time objects occurring in policies. When developing *AuthV2*$_{\text{java}}$, we made the decision to drop millisecond resolution in Date and Time objects to simplify the specification and proofs. Our thinking here was that cloud is a globally distributed system and policy propagation operations can take seconds to complete, making the need for millisecond resolution unnecessary. However, *AuthV1* retains millisecond-level precision and under the right circumstances this can, and did, result in different authorization results than *AuthV2*$_{\text{java}}$. Other mismatches were related to corner cases of wildcard matching and unicode string representation. We fixed all mismatches we encountered as we found them.

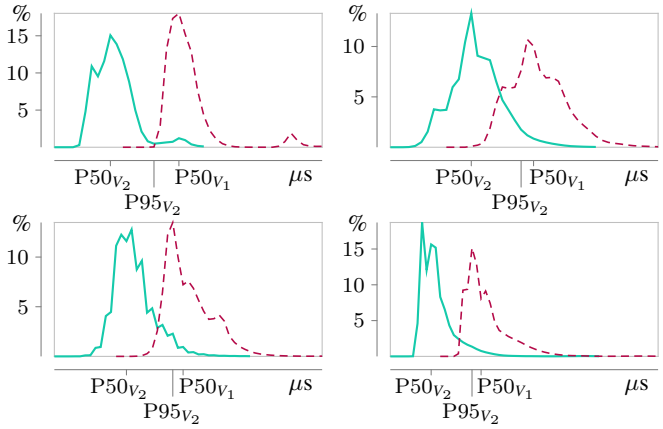

Figure 3: Authorization latency histograms of 4 high-traffic services for 1-week periods before and after the switch to *AuthV2*. The graphs show the performance improvement provided by *AuthV2* across various workloads. The dashed red graph represents *AuthV1*, the solid green one represents *AuthV2*. The x-axis is log-scaled and shows the latency, with the median for each engine, and P95 for *AuthV2*, highlighted. The y-axis shows the percentage of calls at a latency (range). The long tails of the graphs are trimmed at P99.9.

We issued 8 shadow iterations of *AuthV2*$_{\text{java}}$, each a refinement of the previous. The final iteration of *AuthV2*$_{\text{java}}$ agreed with *AuthV1* on 10^{15} inputs with no observed differences. At this point we launched *AuthV2*$_{\text{java}}$ as the production authorization engine and ran *AuthV1* as a shadow to detect any differences that might have evaded our previous shadow testing. After a month of running *AuthV1* as a shadow, and finding no differences, we turned it off completely.

B. Performance testing

A key goal was to ensure that *AuthV2*$_{\text{java}}$'s performance met or exceeded the performance of *AuthV1*, both in terms

of *average latency*, reflecting per-request costs, and *high percentile latencies* (P99, P99.9), reflecting worst-case costs.

While we got some sense for $AuthV2_{\text{java}}$'s performance using synthetic data during development, we did not get the full picture until we did shadow testing. In particular, we found that service workloads varied a fair bit, where this difference owed to their different types of policies and data, and to their different deployment environments. These differences could, in turn, introduce threshold effects owing to just-in-time compilation, garbage collection, and CPU load. Measuring performance during shadow testing gave us a more accurate picture of all this diversity.

Shadow testing also gave us an opportunity to use our formal methods-driven development process to *improve* performance. We started from an improved baseline after the initial rewrite of *AuthV1*, but noticed some regressions (or room for improvement) in corner cases. In response, we optimized the *AuthV2* code, proved it correct, and then deployed the change during the next shadow testing iteration. We never, during this process, observed a correctness-related difference owing to an optimization. Examples of optimizations include removing redundant computations, specializing standard library methods on a subset of input data, and leveraging heap predicates to avoid copying data when it is not mutated.

While we cannot share absolute performance metrics, Fig. 3 provides latency histograms for four high-traffic services on both *AuthV1* and *AuthV2*. On these four services, average latency improved by 69%, P99 by 83% and P99.9 by 85%. Moreover, *AuthV2*'s P95 latency is always less than *AuthV1*'s P50 latency. These improvements are fairly representative of the improvements across all services, which showed a 65% latency improvement, on average.

VIII. Discussion and Related Work

The ideas of formal software verification [30], [56], as well as their ideals and criticisms (see, e.g., Hoare [39], De Millo et al. [21], and Fetzer [28]), are many decades old. Mechanical tools that aid in the process also had an early start [9], [50], followed by specification facilities being directly integrated into programming languages [54] or associated modeling languages [10], [33].

The last fifteen years have seen a major expansion of both formal verification technology and the use of that technology in practice [44]. Several substantial software systems have been verified, including operating systems (OS) kernels like seL4 [37] and FreeRTOS [14]; compilers like CompCert [49] and CakeML [45]; cryptography components or libraries like OpenSSL HMAC [6], EverCrypt [60], Erbsen et al. [27], Almeida et al. [2], and the AWS Encryption SDK (ESDK) [3]; and communication or routing protocols or components thereof, like QUIC [23], SCION [59], and EverParse [62], [67]. All of this software has seen at least some industrial adoption, with the most noteworthy perhaps being the EverCrypt libraries, which are deployed in Firefox and the Linux kernel, both of which are highly security sensitive.

These efforts have leveraged a variety of frameworks to yield proofs of thread and memory safety, functional correctness, and properties about security. Generally speaking, there are basically three development approaches:

1) Write the code in a mainstream language, and use a tool to prove properties specified in the language itself and/or as annotations.
2) Write the code in a mainstream language, and use a separate framework to specify and prove properties about an *embedding* of that program in the framework.
3) Write the code in a *proof oriented programming language* (PPL), a special-purpose language in which the code, specification, and proofs can be expressed. The PPL code is made executable by stripping its logical annotations and compiling it (often in a lightweight manner) to an executable target language.

To support the first approach, researchers have built verifiers for programs written in languages such as C (Frama-C [5], VeriFast [42], and VCC [16]), Java (OpenJML [17] and KeY [1]), Go (Gobra [69]), Ada (SPARK [11]), Rust (Prusti [4], Creusot [24], and Verus [46]), and several others. These tools often work by translating program statements and properties to prove into logical formulae whose truth can be determined by an SMT solver. Building these tools to be precise at scale is challenging: mainstream languages are rich in features, including challenging ones such as mutation, manual memory management, and multi-threading. Of the projects listed above, the SCION Internet router comes closest to taking this approach; it used Isabelle to verify the correctness of the SCION protocol, and then used Gobra to verify that the Go code correctly implements the protocol. One particular challenge for us is that verification tools for Java (KeY and OpenJML) consume specifications written in JML, which does not offer the flexibility we needed to both execute specifications (to test against a legacy implementation) and state and prove properties about the specifications. Other tools also have non-executable specification languages, to varying degrees, and so also suffer this limitation.

To support the second approach, one must express the syntax and semantics of the mainstream language using the language of a proof assistant, such as Isabelle [57], the Rocq prover [18], or Lean [22]. This approach allows specifying properties using the full power of the proof assistant's language, and proving them interactively and/or with automation. Of the above-listed verified software, the seL4, FreeRTOS, and OpenSSL HMAC developments are examples of this approach (all of them considered C programs, mechanized in Isabelle for the first two and the Rocq prover for the last). This approach is flexible and appealing: Rather than internally compiling from source to intermediate language, it essentially exposes the intermediate language as the object of (interactive) proof, leveraging a highly engineered proof environment. That said, it still suffers in that the full source language could be too richly featured for feasible proofs. Another downside is that this can be difficult from an engineer's perspective: They have to

work with programs as embedded terms in the proof assistant, rather as programs in their native syntax with IDE support for definition lookup, code highlighting, etc.

The third approach carries out proof on a program written directly in a PPL, but compiles that program to a mainstream language. CompCert, Erbsen et al., ESDK, and EverCrypt/EverParse are examples of this approach, with code written in the Rocq prover's programming language Gallina, Dafny, and the Low* [61] sublanguage of F* [31]. An advantage of these languages is that they were designed for proof from the start, so while they have many familiar constructs, they also specifically elide features to make proofs easier to construct (whether manually or automatically).

A key challenge of the third approach is compiling to efficient, debuggable code. We met this challenge by writing an idiomatic compiler for DafnyLite, a subset of Dafny we defined. Previously, the F* team developed Low*, a subset of full F* that can be compiled to C using a compiler called KaRaMeL [32]. Compared to our compiler, the goal is the same—intuitive compilation from the limited source to the target language—but the details are quite different. C has manual memory management, undefined behavior, etc. which imposes greater limitations on the source language, and because EverCrypt has few library dependencies, KaRaMeL has little need to deal with an extensive environment model. In contrast, our DafnyLite compiler maps Dafny's classes, inheritance, and error handling as well as our modeled Java environment into idiomatic counterparts in Java. Our compiler is also unique in its use of translation validation via the Java Checker Framework.

Our work is unusual in that we developed a formally verified version of a bespoke legacy system in (highly) active operation. Most of the other work discussed above focused on verifying the implementation of a published standard, e.g., for a language compiler or a cryptographic algorithm. Thus, they had more reliable and meticulous documentation from which to develop the specification, and clearer properties to prove. For us, extensive property-based and differential testing, including against production data, was critical to ensure our specification accurately captured the expected behavior and achieved good performance.

Another authorizer developed recently with the help of formal methods is Cedar [19]. Rather than develop a proved-correct implementation of Cedar via one of the three approaches discussed above, the Cedar developers took a verification-*guided* approach [26] comprising three parts: 1) hand-write and prove key properties on an executable specification of Cedar in the Lean verification-aware language [22]; 2) hand-write an implementation of Cedar in Rust; and 3) use differential random testing [52] to assure that the Lean specification and Rust implementation behave the same. Verfication-guided development resembles the approach we took with *AuthV2* but instead of *compiling* a *proved-correct* Lean implementation to Rust, as we compiled a proved-correct DafnyLite implementation to Java, the Cedar developers *directly wrote* the Rust implementation and *tested* its correctness against the Lean specification. This means *AuthV2* offers comparatively higher assurance that its implementation $AuthV2_{\text{java}}$ truly matches its specification $AuthV2^{\text{spec}}_{\text{dafny}}$. This assurance relies on the assumption that the DafnyLite compiler produces a correct result, whereas Cedar relies on the correctness of the Lean compiler, since the Lean spec is compiled to C in order to differentially test it. We gained substantial assurance in the DafnyLite compiler's correctness on *AuthV2* through extensive shadow testing of $AuthV2_{\text{java}}$ against *AuthV1*, through the use of the Checker framework, and by being able to manually review the generated code (which is quite similar to the Dafny original). Another difference between Cedar and *AuthV2* is that Cedar was not a legacy system, so it did not involve the same reverse-engineering challenges or the far more extensive testing needed for ensuring compatibility and stakeholder buy-in that *AuthV2*'s development involved.

IX. Conclusion

In this paper, we presented a major success story to add to the catalog of formal method "wins." Thirty years ago, formal methods were still regarded as a technological outsider: too difficult, too limited, too expensive, and generally unnecessary. The strongest evidence for this is the somewhat defensive stance of its proponents [8], [34]. Since then, tools and techniques improved significantly, and combined with advances in computing power, the gap between theory and applications has gradually closed. This has led to broader adoption and consequently many more successes [38], [47], [68].

We shared our experience formally specifying and replacing a long-running, heavily utilized, correctness- and security-critical piece of legacy software—the AWS authorization engine. This experience included overcoming logical, technical, and social challenges. Our effort has resulted in one of the most thoroughly analyzed pieces of software at Amazon.

We employed a three-pronged approach that combines formal verification, human code review, and systematic large-scale testing. It required three handwritten implementations, the development of an idiomatic compiler, rigorous manual reviews of the output by our team and stakeholders at every stage, and comprehensive testing approaches such as fuzzing, regression testing, and shadow-mode deployment. The benefits are clear: we can now confidently make changes and introduce new features to the software swiftly, while preserving robustness, performance, and backward compatibility.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification — The KeY Book — From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, 2020.

[3] Amazon Web Services. AWS Encryption SDK for Dafny, 2024. https://github.com/aws/aws-encryption-sdk-dafny.

[4] Vytautas Astrauskas, Aurel Bílỳ, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The Prusti project: Formal verification for Rust. In *NASA Formal Methods Symposium*. Springer, 2022.

[5] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: The Frama-C software analysis platform. *Communications of the ACM*, 64(8):56–68, 2021.

[6] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of OpenSSL HMAC. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*, pages 207–221, Washington, D.C., 2015. USENIX Association. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer.

[7] Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. Challenges and experiences in managing large-scale proofs. In *Proceedings of the 11th Conference on Intelligent Computer Mathematics (CICM '12)*, pages 32–48, Berlin, Heidelberg, 2012. Springer Verlag. https://www.tbrk.org/papers/cicm2012.pdf.

[8] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995. https://www.site.uottawa.ca/~bochmann/CSI5174/CourseNotes/Literature/Bowen%20-%207%20more%20myths.pdf.

[9] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, Inc., 1979. https://www.cs.utexas.edu/~boyer/acl.pdf.

[10] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005. https://homes.cs.washington.edu/~mernst/pubs/jml-tools-sttt2005.pdf.

[11] Roderick Chapman, Claire Dross, Stuart Matthews, and Yannick Moy. Co-developing programs and their proof of correctness. *Communications of the ACM*, 67(3):84–94, 2024.

[12] Roderick Chapman and Florian Schanda. Are we there yet? 20 years of industrial theorem proving with SPARK. In *Interactive Theorem Proving*, pages 17–26, Berlin, Heidelberg, 2014. Springer Verlag. https://proteancode.com/keynote.pdf.

[13] Checker Framework. The checker framework, 2024. https://checkerframework.org/.

[14] Nathan Chong and Bart Jacobs. Formally verifying FreeRTOS' interprocess communication mechanism. Embedded World Exhibition & Conference 2021, 2021. https://www.amazon.science/publications/formally-verifying-freertos-interprocess-communication-mechanism.

[15] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP '00)*, pages 268–279, New York, NY, 2000. Association for Computing Machinery. https://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf.

[16] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: a practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[17] David R. Cok, Gary T. Leavens, and Mattias Ulbrich. Java Modeling Language (JML) reference manual, 2024. https://www.openjml.org/.

[18] The Coq Development Team. The Coq proof assistant, 2024. https://coq.inria.fr.

[19] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew M. Wells. Cedar: A new language for expressive, fast, safe, and analyzable authorization. *Proc. ACM Program. Lang.*, 8(OOPSLA1), April 2024.

[20] The Dafny Community. The Dafny programming language, 2024. https://dafny.org/.

[21] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[22] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction — CADE 28*, pages 625–635, Berlin, Heidelberg, 2021. Springer Verlag. https://pp.ipd.kit.edu/uploads/publikationen/demoura21lean4.pdf.

[23] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. A security model and fully verified implementation for the IETF QUIC record layer. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP '21)*, pages 1162–1178, Washington, DC, 2021. IEEE Computer Society. https://www.microsoft.com/en-us/research/publication/security-model-verified-implementation-quic-record-layer/.

[24] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for the deductive verification of Rust programs. In *Proceedings of the 23rd International Conference on Formal Engineering Methods (ICFEM)*. Springer-Verlag, 2022.

[25] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, 2011.

[26] Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, John Kastner, Anwar Mamat, Matt McCutchen, Neha Rungta, Bhakti Shah, Emina Torlak, and Andrew Wells. How we built cedar: A verification-guided approach. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 351–357, New York, NY, USA, July 2024. Association for Computing Machinery.

[27] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic: With proofs, without compromises. *ACM SIGOPS Operating Systems Review*, 54(1):23–30, 2020.

[28] James H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31(9):1048–1063, 1988. https://dl.acm.org/doi/10.1145/48529.48530.

[29] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, 2016. https://hal.science/hal-01396864v1.

[30] Robert W. Floyd. Assigning meanings to programs. *Proceedings of the Symposium on Applied Mathematics*, 19:19–31, 1967. https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf.

[31] The F* Development Team. F*: A proof-oriented programming language, 2024. https://www.fstar-lang.org.

[32] The F* Development Team. Fstarlang/karamel: KaRaMeL is a tool for extracting low-level F* programs to readable C code. https://github.com/FStarLang/karamel, 2024. .

[33] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Spring-Verlag, 1993. https://www.cs.cmu.edu/afs/cs/usr/wing/www/publications/LarchBook.pdf.

[34] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990. https://www.rose-hulman.edu/class/cs/cs415/Examples/hall7myths.pdf.

[35] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael Lowell Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: Proving safety and liveness of practical distributed systems. *Communications of the ACM*, 60(7):83–92, 2017.

[36] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14*, pages 165–181. USENIX Association, 2014.

[37] Gernot Heiser, Gerwin Klein, and June Andronick. seL4 in Australia: From research to real-world trustworthy systems. *Communications of the ACM*, 63(4):72–75, 2020. https://trustworthy.systems/publications/csiro_full_text/Heiser_KA_20.pdf.

[38] Michael G. Hinchey and Jonathan P. Bowen. *Industrial-strength formal methods in practice*. Springer Science & Business Media, 2012.

[39] C. A. R. Hoare. The mathematics of programming. In *Foundations of Software Technology and Theoretical Computer Science*, pages 1–18, Berlin, Heidelberg, 1985. Springer Verlag. https://ora.ox.ac.uk/objects/uuid:dcd96ff3-8fd9-4412-8733-892641107d63/files/m88f2b80f31ed89dead6c09c65011ce07.

[40] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. *ACM SIGPLAN Notices*, 47(10):879–896, 2012.

[41] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamaric, Neha Rungta, and Emina Torlak. Testing Dafny (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*, pages 556–567, New York, NY, 2022. Association for Computing Machinery. https://www.amazon.science/publications/testing-dafny-experience-paper.

[42] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.

[43] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating-system kernel. *Communications of the ACM*, 53(6):107–115, 2010.

[44] Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby Murray, and Gernot Heiser. Formally verified software in the real world. *Communications of the ACM*, 61(10):68–77, 2018. https://www.trustworthy.systems/publications/csiro_full_text/Klein_AKMHF_18.pdf.

[45] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[46] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023.

[47] Thierry Lecomte. Applying a formal method in industry: A 15-year trajectory. In *Formal Methods for Industrial Critical Systems*, pages 26–34. Berlin, Heidelberg, 2009. https://link.springer.com/chapter/10.1007/978-3-642-04570-7_3#citeas.

[48] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 14th International conference on logic for programming artificial intelligence and reasoning (LPAR '10)*, pages 348–370, Berlin, Heidelberg, 2010. Springer Verlag. https://www.microsoft.com/en-us/research/publication/dafny-automatic-program-verifier-functional-correctness-2/.

[49] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert — a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, 2016. SEE. https://hal.inria.fr/hal-01238879.

[50] D. C. Luckham, S. M. German, F. W. v. Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier: User manual. Stanford Verification Group, 1979. http://i.stanford.edu/pub/cstr/reports/cs/tr/79/731/CS-TR-79-731.pdf.

[51] Christopher A. Mackie. Preventing signedness errors in numerical computations in Java. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1148–1150, 2016.

[52] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[53] Sean McLaughlin, Georges-Axel Jaloyan, Tongtong Xiang, and Florian Rabe. Enhancing proof stability. In *Proceedings of the 2024 Dafny Workshop (POPL)*. Association for Computing Machinery, 2024. https://popl24.sigplan.org/details/dafny-2024-papers/14/Enhancing-Proof-Stability.

[54] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice Hall International, 1988. https://bertrandmeyer.com/wp-content/upLoads/OOSC2.pdf.

[55] Microsoft. Shadow testing, 2024. https://microsoft.github.io/code-with-engineering-playbook/automated-testing/shadow-testing/.

[56] F. L. Morris and C. B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, 1984.

[57] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A proof assistant for higher-order logic*. Springer Verlag, Berlin, Heidelberg, 2021. https://isabelle.in.tum.de/doc/tutorial.pdf.

[58] Paul R. Holser, Jr. junit-quickcheck: Property-based testing, junit-style, 2010. https://pholser.github.io/junit-quickcheck/.

[59] João C. Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix A. Wolf, Marco Eilers, Christoph Sprenger, David Basin, Peter Müller, and Adrian Perrig. Protocols to code: Formal verification of a next-generation Internet router. Technical report, arXiv, 2024.

[60] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP '20)*, pages 983–1002, Washington, DC, 2020. IEEE Computer Society. https://www.microsoft.com/en-us/research/publication/evercrypt-a-fast-veri%EF%AC%81ed-cross-platform-cryptographic-provider/.

[61] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Want, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. *Proceedings of the ACM on Programming Languages*, 1(ICFP), 2017. https://hal.archives-ouvertes.fr/hal-01672706.

[62] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified secure zero-copy parsers for authenticated message formats. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*, pages 1465–1482, Santa Clara, CA, 2019. USENIX Association. https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud.

[63] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. Towards making formal methods normal: Meeting developers where they are. In *Presented at the HATRA 2020 workshop*, 2020. https://arxiv.org/abs/2010.16345.

[64] Suhrid Satyal, Ingo Weber, Hye-young Paik, Claudio Di Ciccio, and Jan Mendling. Shadow testing for business process improvement. In *On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part I*, volume 11229 of *Lecture Notes in Computer Science*, pages 153–171, 2018.

[65] Gerald Schermann, Jürgen Cito, and Philipp Leitner. Continuous experimentation: Challenges, implementation techniques, and current research. *IEEE Software*, 35(2):26–31, 2018.

[66] Amazon Web Services. IAM policy elements: Condition operators, 2024. https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements_condition_operators.html.

[67] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irinia Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vasquew, Michael Tang, Omar Cardona, and Arti Gupta. Hardening attack surfaces with formally proven binary format parsers. In *Proceedings of the 43rd AC: SIGLAN International Conference on Programming Language Design and Implementation (PLDI '22)*, pages 31–45, New York, NY, 2022. Association for Computing Machinery. https://www.microsoft.com/en-us/research/publication/hardening-attack-surfaces-with-formally-proven-binary-format-parsers/.

[68] Benjamin Weyers, Michael D. Harrison, Judy Bowen, Alan J. Dix, and Philippe A. Palanque. Case studies. In *The Handbook of Formal Methods in Human-Computer Interaction*, pages 89–121, Berlin, Heidelberg, 2017. Springer Verlag. https://link.springer.com/chapter/10.1007/978-3-319-51838-1_4.

[69] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. Gobra: Modular specification and verification of Go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification (CAV)*, volume 12759 of *Lecture Notes in Computer Science*, pages 367–379. Springer, 2021.

[70] Yi Zhou, Jay Bosamiya, Jessica Li, Marijn Heule, and Bryan Parno. Context pruning for more robust SMT-based program verification. In

Proceedings of the Formal Methods in Computer-Aided Design (FMCAD) Conference, October 2024.

[71] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. Mariposa: Measuring smt instability in automated program verification. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD) Conference*, 2023.