Project Everest: Perspectives from Developing Industrial-Grade High-Assurance Software

DANEL AHMAN, University of Tartu, Tartu, Estonia
KARTHIKEYAN BHARGAVAN, INRIA, Le Chesnay, France
BARRY BOND, Self Employed, Redmond, Washington, USA
JAY BOSAMIYA, Microsoft Research, Redmond, Washington, USA
CHRISTOPHER BRZUSKA, Aalto University, Aalto, Finland
ANTOINE DELIGNAT-LAVAUD, Microsoft UK Ltd—Reading, Cambridge, United Kingdom of Great Britain and Northern Ireland
CÉDRIC FOURNET, Microsoft UK Ltd.—Reading, Cambridge, United Kingdom of Great Britain and Northern Ireland
AYMERIC FROMHERZ, INRIA, Paris, France
SYDNEY GIBSON, ZeroRISC, Pittsburgh, Pennsylvania, USA
CHRIS HAWBLITZEL, Microsoft Research, Redmond, Washington, USA
CĂTĂLIN HRIȚCU, MPI-SP, Bochum, Germany
MARKULF KOHLWEISS, University of Edinburgh, Edinburgh, United Kingdom of Great Britain and Northern Ireland
GUIDO MARTÍNEZ, Microsoft Research, Redmond, Washington, USA
HAOBIN NI, University of Washington, Seattle, Washington, USA
BRYAN PARNO, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
JONATHAN PROTZENKO, Microsoft Corporation, Redmond, Washington, USA
TAHINA RAMANANANDRO, Microsoft Research, Redmond, Washington, USA
ASEEM RASTOGI, Microsoft Research India, Bengaluru, India
EXEQUIEL RIVAS, Tallinn University of Technology, Tallinn, Estonia
NIKHIL SWAMY, Microsoft Research, Redmond, Washington, USA
SANTIAGO ZANELLA-BÉGUELIN, Microsoft UK Ltd.—Reading, Cambridge, United Kingdom of Great Britain and Northern Ireland

Authors' Contact Information: Danel Ahman, University of Tartu, Tartu, Estonia; e-mail: danel.ahman@ut.ee; Karthikeyan Bhargavan, INRIA, Le Chesnay, France; e-mail: Karthikeyan.Bhargavan@inria.fr; Barry Bond, Self Employed, Redmond, Washington, USA; e-mail: RedmondBarryBo@outlook.com; Jay Bosamiya, Microsoft Research, Redmond, Washington, USA; e-mail: jayb@microsoft.com; Christopher Brzuska, Aalto University, Aalto, Finland; e-mail: chris.brzuska@gmail.com; Antoine Delignat-Lavaud, Microsoft UK Ltd—Reading, Cambridge, United Kingdom of Great Britain and Northern Ireland; e-mail: antdl@microsoft.com; Cédric Fournet, Microsoft UK Ltd.—Reading, Cambridge, United Kingdom of Great Britain and Northern Ireland; e-mail: fournet@microsoft.com; Aymeric Fromherz, INRIA, Paris, France; e-mail: aymeric.fromherz@inria.fr; Sydney Gibson, ZeroRISC, Pittsburgh, Pennsylvania, USA; e-mail: sydgibs@gmail.com; Chris Hawblitzel, Microsoft Research, Redmond, Washington, USA; e-mail: chrishaw@microsoft.com; Cătălin Hriţcu, MPI-SP, Bochum, Germany; e-mail: catalin.hritcu@mpi-sp.org; Markulf Kohlweiss, University of Edinburgh, Edinburgh, United Kingdom of Great Britain and Northern Ireland; e-mail: markulf@outlook.com; Guido Martínez, Microsoft Research, Redmond, Washington, USA; e-mail: guimartinez@microsoft.com; Haobin Ni, University of Washington, Seattle, Washington, USA; e-mail: haobin.ni@gmail.com; Bryan Parno, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; e-mail: bparno@andrew.cmu.edu; Jonathan Protzenko, Microsoft Corporation, Redmond, Washington, USA; e-mail: protz@microsoft.com; Tahina Ramananandro, Microsoft Research, Redmond, Washington, USA; e-mail: taramana@microsoft.com; Aseem Rastogi, Microsoft Research India, Bengaluru, India; e-mail: aseemr@microsoft.com; Exequiel Rivas, Tallinn University of Technology, Tallinn, Estonia; e-mail: exequiel.rivas@ttu.ee; Nikhil Swamy (corresponding author), Microsoft Research, Redmond, Washington, USA; e-mail: nswamy@microsoft.com; Santiago Zanella-Béguelin, Microsoft UK Ltd.—Reading, Cambridge, United Kingdom of Great Britain and Northern Ireland; e-mail: santiago@microsoft.com.

ACM 1558-4593/2026/6-ART9
https://doi.org/10.1145/3805702

Project Everest began at Microsoft Research in 2016, aiming to spur research in program verification to produce industrial-grade software. In collaboration with INRIA and Carnegie Mellon University, Project Everest's goal was to produce drop-in verified replacements of secure communications software used in the HTTPS ecosystem, including TLS, the underlying cryptography, and related subprotocols. Now, almost a decade later, we reflect on the project, sharing both its successes and failures, and look ahead to the next decade of program verification research.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Security and privacy → Logic and verification**;

Additional Key Words and Phrases: Program proof, Cryptographic software, Systems software, Communications software, Proof-oriented programming

ACM Reference format:
Danel Ahman, Karthikeyan Bhargavan, Barry Bond, Jay Bosamiya, Christopher Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Aymeric Fromherz, Sydney Gibson, Chris Hawblitzel, Cătălin Hriţcu, Markulf Kohlweiss, Guido Martínez, Haobin Ni, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Exequiel Rivas, Nikhil Swamy, and Santiago Zanella-Béguelin. 2026. Project Everest: Perspectives from Developing Industrial-Grade High-Assurance Software. *ACM Trans. Program. Lang. Syst.* 48, 2, Article 9 (June 2026), 64 pages.
https://doi.org/10.1145/3805702

1 Introduction

Project Everest began in 2016 at the instigation of Jeannette Wing, then director of Microsoft Research (MSR). Inspired by the *Expeditions in Computing*[1] program that she had helped create at the National Science Foundation, Wing sought to spur ambitious, multi-year industrial research programs by creating an Expeditions program at MSR. An internal call for proposals was issued, with funded proposals supported for 3–5 years by MSR, above and beyond regular support, with dedicated funds for additional interns, developers, facility needs, and visitors.

In the second year of the MSR Expeditions program (for 2016), four of the present authors (Fournet, Hawblitzel, Parno, and Swamy) submitted a proposal titled *Deploying a Verified Secure Implementation of the HTTPS Ecosystem.* The HTTPS ecosystem was broadly recognized as crucial to Internet security, and a series of high-profile attacks had raised awareness of the underlying flaws both in the design of the core TLS [182] protocol (c.f., attacks such as LogJam[2]) and in widely used implementations (c.f., attacks like Heartbleed[3] and Apple's GOTO Fail[4]). MSR had deep expertise in cryptography and security, as well as program and protocol verification, spread across several labs, and we saw producing verified implementations of components of HTTPS as an opportunity to work together on an ambitious, impactful goal. Additionally, a new version of the protocol, TLS-1.3, was being drafted then at the IETF, and we anticipated that working on verifying the protocol could help improve the standardization process and also help software teams that would need to implement the new standard.

Our proposal was approved and funded in 2016. It was initially named ImPS (for Impervious Protocol Security) but by the spring of 2016 it had been renamed to Everest, a recursive acronym: Everest Verified End-to-end Secure Transport. We had milestones set for a five-year period, including both research outcomes as well as anticipated deployments. Figure 1 shows the components we were

[1] https://www.nsf.gov/cise/ccf/expeditions-awards.
[2] https://weakdh.org/.
[3] https://heartbleed.com/.
[4] https://www.imperialviolet.org/2014/02/22/applebug.html.

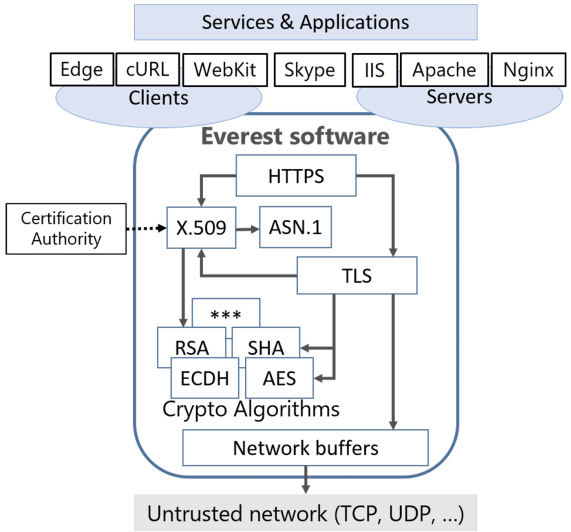

Fig. 1. Architecture of proposed Everest software, circa 2016.

targeting. Of course, the trajectory of the project evolved significantly as it progressed, something we reflect on throughout this article.

In a 2017 paper [36], we described Project Everest as "*a new joint project between Microsoft Research and INRIA that aims to build verified software components and deploy them within the existing software ecosystem.*" At that point, although research in software verification had progressed sufficiently to enable the development of substantial end-to-end verified artifacts [39, 102, 108, 109, 126, 142, 194], few artifacts had reached the maturity and modularity needed to enable commercial deployment. By targeting the HTTPS ecosystem, we believed we could provide "*a large boost to software security*" by developing high-assurance components that could be deployed piecemeal, rather than requiring a wholesale replacement of the software stack, as would be required if one were to offer, say, a verified operating system.

However, as the project name "Everest" suggests, we recognized that producing a verified implementation of TLS and all its associated subprotocols and algorithms would require a monumental effort.[5] At its inception in 2016, the project was slated to take 5 years and already included a large team with expertise in programming languages, program verification, systems, security, and cryptography. We had 21 researchers and engineers at Microsoft Redmond and Cambridge, three from INRIA (Paris), and one from LRI (Orsay), indicating their interest to participate at different levels of involvement.

Eventually, the team grew to be distributed across three institutions in five main locations: Microsoft Research (Redmond, Cambridge, and Bangalore), Carnegie Mellon University (Pittsburgh), and INRIA (Paris), along with many contributors from other places, including Edinburgh, Scotland; Rosario, Argentina; Bochum, Germany; and Tallinn, Estonia. In addition, many interns and PhD students contributed substantially to the work. Our collaboration was enabled by an open source model, with all development on GitHub, with all stakeholders recognizing the importance of

[5] Amusingly, Google's Project Wycheproof (https://github.com/C2SP/wycheproof), a security testing suite for cryptographic primitives, is named after Mount Wycheproof, sometimes called "the smallest mountain in the world." Their motto is: "The smaller the mountain the easier it is to climb it!" a tongue-in-cheek reference to Project Everest's opposite approach.

conducting security research in the open and inviting public scrutiny. A full list of contributors is in Appendix A.

Geographic distribution was a challenge which we tackled, in part, by gathering every 6 months in either Europe or the United States for an in-person "all-hands" meeting, which was very useful for cohesion, planning, and building team spirit. However, starting with the pandemic in 2020, these in-person gatherings ceased and the project became less close-knit. We also used a Slack instance for most team communications, using separate channels for each sub-project; membership grew to several hundred members in total. Subsequently, many of these sub-projects migrated to their own public discussion forums on Zulip.

The core of Project Everest was, roughly, the fusion of two teams: security and systems researchers who had been using Dafny [140] as part of the Ironclad [109] and IronFleet [108] projects; and programming languages and security researchers who had been developing F⋆ [200] and using it for the miTLS [39] project. Ironclad was recognized as a significant milestone in developing verified systems and included a small but complete end-to-end verified stack, including an OS kernel; drivers; system and crypto libraries including SHA, HMAC, and RSA; and four "Ironclad apps." Further, IronFleet proved the safety and liveness of several complex distributed protocol implementations, including Paxos [134]. miTLS was also well recognized in the research community, with some its authors winning the Levchin Prize for it in 2016, for analysis of TLS-1.2 and developing tools for formal proofs of protocols.[6]

We debated at length the choice of language and tools, which we summarize in Section 2.1.1. The plan we settled on involved developing verified code in F⋆, a proof-oriented programming language, together with several **domain-specific languages (DSLs)** embedded in F⋆. A project of this scale had not yet been attempted in F⋆, so we anticipated enhancing the language and its tooling as we went along. We planned to prove all of the code at least functionally correct in F⋆, and then extract it to existing languages, like C and assembly, for integration within existing software projects. The verified code itself was to include the following:

(1) Cryptographic primitives and constructions, in C and assembly, including support for all the main cipher suites used in TLS, with performance comparable to existing, unverified implementations.
(2) The ASN.1 data description language [118] and, specifically, support for X.509 public key certificates [64], as well as the code necessary to validate X.509 certificate chains.
(3) The TLS protocol, including support for both TLS-1.2 and TLS-1.3, covering both the record-layer and handshake subprotocols, with idealized code serving as a cryptographic specification.
(4) The HTTPS protocol, particularly the many subtle web security issues around it [7, 58], e.g., for cookie management and content security policies.

Looking back, we accomplished much (but not all) of what we set out to do, though, as may be expected, the project's course and goals were revised multiple times. Most significantly, we did not tackle the HTTPS protocol itself. Rather, we focused on the layers beneath it, including TLS and also QUIC [135]. While we did offer fully verified, high-performance implementations of the record-layer protocols for TLS and QUIC, we did not complete verifying our *implementation* of the TLS handshake, though we did produce a detailed pen-and-paper proof of its cryptographic security, developing a novel technique for cryptographic proofs. On the other hand, we spent a significant effort working on things that were not in our original project plan, reacting to various opportunities and changes in the landscape as they occurred (e.g., QUIC).

[6] https://rwc.iacr.org/LevchinPrize/winners.html#mitls.

Figure 2 summarizes our various contributions and the relationships among them. At the time of writing, a full build of Project Everest takes about 2 hours on a build server with 24 threads, building repositories with a total of about 1.3 million lines of non-auto-generated F* source text. The build issues around 66,000 SMT queries to automatically discharge around 622,000 separate proof obligations, producing around 4,200 checked F* files. We do not have an easy way to count proof obligations discharged without the use of an SMT solver, e.g., proofs by unification, normalization, or tactics. The overall proof-to-code ratio is highly variable—some projects do not produce any verified executable code, instead offering verification libraries only; others offer code generators that automatically produce large amounts of verified code from specifications. The largest project with human-authored source extracted to executable code, HACL*, contains about 245,000 lines of F* and Vale source files, and yields about 135,000 lines of verified C and assembly code.

Our contributions can be grouped into four areas: verification tools and methodologies; cryptographic algorithms; data formats and parsing; protocol modeling and verification. In spanning these areas, Project Everest was perhaps unique in its co-development of (i) verification tools and methodologies, (ii) modeling and verification of complex systems and their properties, and (iii) actual deployment of verified software. Many offshoots of Project Everest are active today, furthering research in these and other areas.

Verification Tools and Methodologies. Project Everest was instrumental in the development of F*. Indeed, many of the language's features, sub-languages, and libraries were developed specifically to address the needs of Project Everest, and one could reasonably say that the language and its applications were co-designed. F* today supports nearly a dozen DSLs, each catering to a particular kind of programming and program verification task. Tooling surrounding F*, notably the KaRaMeL compiler, extracts code written in low-level F* dialects (specifically, a DSL called Low* [173]) to C (and recently also Rust [94]), and such code is deployed in a variety of commercial products, ranging from Windows to Firefox. The verification tools and methodologies developed by Project Everest are documented in several papers [5, 6, 93, 95, 100, 147–149, 173, 202].

Post-Everest. Today, more than a million lines of code and proof are maintained in F*'s **continuous integration (CI)** pipeline, even as the language continues to evolve. Further, lessons learned from using and developing F* inform newer verification tools, including Pulse [80] for proofs in **concurrent separation logic (CSL)**, and Hax [38], Verus [136, 137] and Aeneas [111], which target the verification of Rust programs.

Cryptographic Primitives. Cryptographic primitives and constructions have abstract mathematical specifications, with security-critical, high-performance implementations in languages like C and assembly. This makes them ideal candidates for verification, and indeed, over the past decade, there has been a flourishing effort across multiple research groups aiming to verify cryptographic primitives [8, 15, 75, 85, 208]. Project Everest produced comprehensive libraries of cryptographic implementations, covering a broad range of cipher suites, in C (HACL*) and in assembly (ValeCrypt), coupled with a cryptographic provider (EverCrypt) providing high-level interfaces suitable for use in a variety of applications. Verified cryptographic implementations from Project Everest are now deployed in many commercial products, including Mozilla Firefox, the Linux kernel, Python, mbedTLS, the Tezos blockchain, the ElectionGuard electronic voting SDK, and the WireGuard VPN. At the time, our implementations of AES-GCM in assembly and Curve25519 in a mixture of C and assembly were the fastest implementations available, verified or not, and they continue to be competitive. The underlying research is documented in several papers [46, 93, 112, 168, 172, 231].

Post-Everest. Lessons learned from our efforts inform new cryptographic verification projects, including notably on post-quantum algorithms [125] and simplifying the verification of cryptographic routines that run on heterogeneous hardware platforms [228].

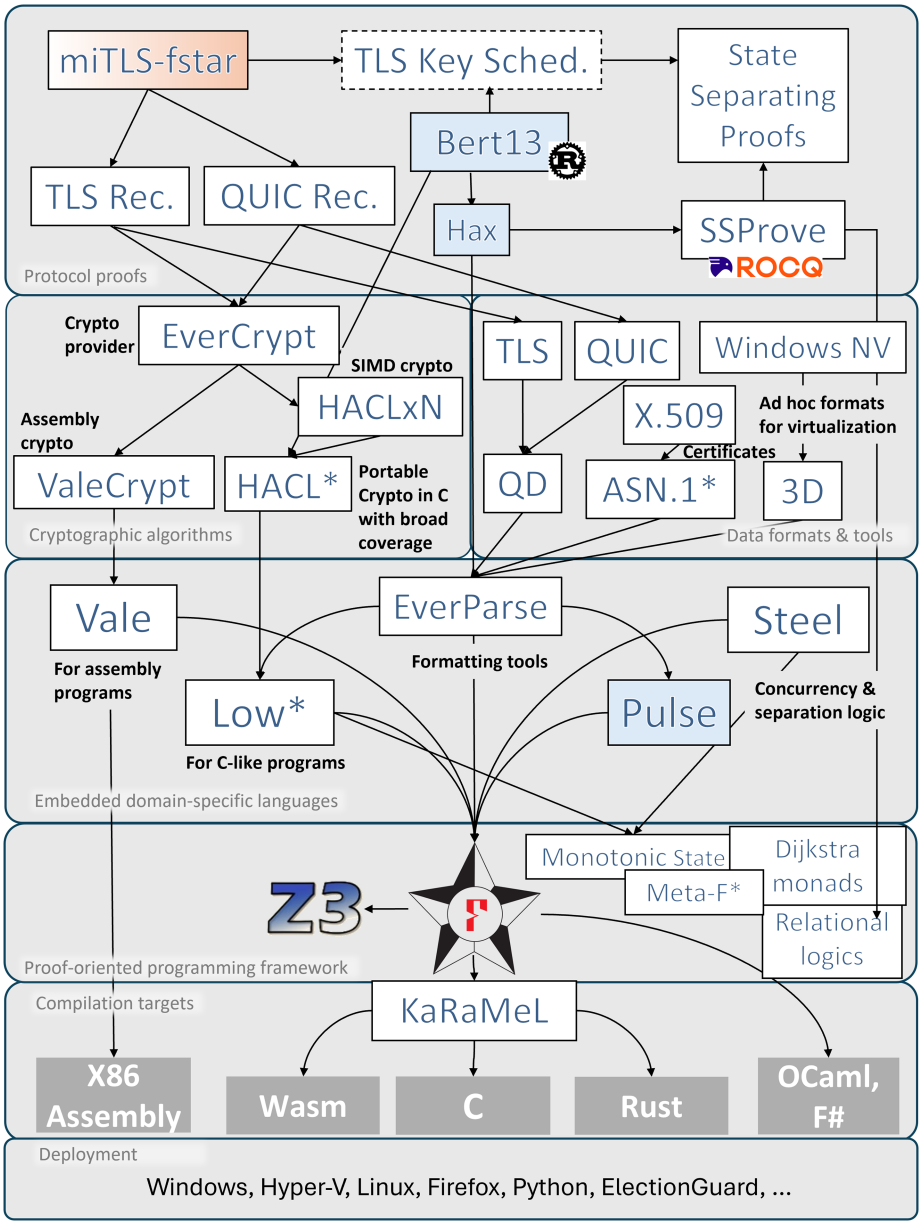

Fig. 2. *The Architecture of Project Everest*, with arrows indicating dependencies among components. Z3 and Rocq were developed independently. All our verified, executable software and their specifications were developed using several DSLs embedded in F⋆. The proof of the TLS-1.3 handshake in miTLS-fstar was left incomplete, and, as such, is shown in a different color, though the underlying key schedule was formalized on paper using the SSPs methodology. We also built SSProve, a tool to conduct SSPs of cryptographic security in Rocq, though by the end of Project Everest, we had not connected SSProve to miTLS-fstar. Many tools and techniques (shown in blue) continue to be developed using parts of Project Everest's framework, e.g., Pulse, Bert13, and Hax. Bert13 is notable in that it yielded a proof of security for TLS-1.3, using SSProve to mechanize Project Everest's TLS key schedule proof, in combination with ProVerif, Hax, and F⋆, for an end-to-end proof of a Rust implementation of the protocol.

Data Formats and Parsing. Processing binary formatted data packets is a routine task in protocol implementations, yet it is easy to get wrong and has been the source of many security critical bugs, including in the HTTP ecosystem [122]. In cryptographic settings, it is also important for the formats themselves to be non-malleable, e.g., an attacker should not be able to alter a binary message without also changing how it is parsed. To process such formats, we developed EverParse [176], a verified parser and serializer generator for non-malleable binary formats. Initially, we used EverParse for TLS protocol messages. We also used EverParse to formalize ASN.1, the language in which X.509 certificates are described, and produced the first proof of security of its **distinguished encoding rules (DER)** [156], though our executable code for ASN.1 was only extracted to OCaml rather than to efficient, low-level C code.

However, the potential uses of EverParse were broader than we expected. Many low-level software components need to parse attacker-controlled binary formatted data, and a tool that generates secure code for this task proved to be more broadly useful. Working closely with engineering teams from Windows Networking, we designed 3D, a language for specifying ad hoc, binary formatted data, and used EverParse to generate verified C code to parse binary messages specified in 3D [201]. This has been particularly useful in Microsoft Azure's network virtualization stack, where since 2021, every network message, on both ingress and egress, has all its encapsulation headers validated and parsed with EverParse, helping to defend against attacks from both network adversaries and potentially malicious virtual machine guests that could otherwise compromise the security and integrity of the host kernel.

Post-Everest. More recently, with learnings from EverParse, the next generation of formatting tools have emerged, including PulseParse [177], a framework building on EverParse in Pulse to handle recursive formats including in standards such as CBOR [47] and CDDL [42] and producing C or Rust code, and Vest [57], an EverParse-inspired generator of fast and secure Rust parsers and serializers verified using Verus.

Protocol Analysis and Verification. Everest's implementation of TLS was based on miTLS (Microsoft-Inria TLS) [39], an executable, verified implementation of TLS-1.2 initially developed in F# and verified using F7 [24], a refinement type checker for F#. However, miTLS was not intended for deployment, being implemented in high-level, purely functional F#. We aimed to re-implement miTLS in Low* with an F* specification, extending it to support both TLS-1.2 and TLS-1.3, and to produce C code, with proofs of functional correctness and cryptographic security, i.e., proving the protocol design and its implementation correct and secure. For clarity, we dub this new version miTLS-fstar.

We spent a considerable effort trying to develop a mechanically checked cryptographic proof of security for the TLS handshake, as implemented in miTLS-fstar. Towards this end, we developed a new methodology for cryptographic proofs called **state-separating proofs (SSP)** [54],[7] using it to develop a detailed pen-and-paper proof of cryptographic security for the TLS handshake, including its main key schedule [53]. We also built SSProve [3], a tool to mechanize SSPs in the Rocq proof assistant. However, due to a variety of factors, including the shifting priorities of a large team, we did not manage to translate this pen-and-paper proof to the verified code of miTLS-fstar. Nevertheless, miTLS-fstar is a complete implementation of TLS-1.3 and interoperates with other implementations. Although it lacked a full proof for the handshake, miTLS-fstar did have a verified, high-performance implementation of the TLS-1.2 and TLS-1.3 record layer protocols [72]—the first such end-to-end proof for those subprotocols.

In addition, during the project, the QUIC protocol [135] emerged as an important variant of TLS running over UDP, and we produced a verified implementation of its record layer as well

[7]The methodology was independently invented by Mike Rosulek and is the basis of his *The Joy of Cryptography* textbook.

[73]. miTLS-fstar's (unverified) handshake can also be used independently for QUIC, together with a small runtime library, and this implementation was used by initial versions of WinQUIC, the Windows implementation of QUIC.

The results of our protocol analysis and implementation efforts also revealed several issues in the TLS-1.3 specification while it was being drafted, and these issues were fixed before the protocol was standardized [182]. The final TLS-1.3 RFC acknowledges the attacks and fixes discovered by several members of Project Everest, including them as contributors and citing several papers.

Post-Everest. Building on lessons learned from Project Everest, the DY* framework [32] embedded in F* was developed to enable modular proofs of cryptographic protocols in the symbolic model. In contrast, the Owl tool [98] provides type-based verification of computational security for protocol designs and automatically produces verified implementations in Rust [192]. These tools have been used to produce security proofs of a range of related protocol standards, including MLS [52, 213], in the computational and symbolic model respectively—Wallez et al. were awarded a prestigious Internet Defense Prize[8] for their work on proving the security of TreeSync, a key sub-protocol of MLS, using F* and DY*. Perhaps most relevant is recent work on Bert13 [41], which builds on Project Everest's tools, techniques, and libraries (including F*, HACL*, SSProve, and our proof of the TLS-1.3 key schedule) to finally complete what was left incomplete in miTLS-fstar, i.e., an end-to-end proof of security for an executable implementation of TLS-1.3. Bert13 is implemented in Rust[9] and uses Hax [38] (cf. Section 5) to translate Rust code to F*, SSProve, and ProVerif for different parts of the proof, including connecting Bert13's key schedule to a mechanized version of our pen-and-paper proof of the TLS-1.3 key schedule in SSProve.

1.1 Outline and Summary of Takeaways

In the rest of this article, we summarize our core research contributions; the technical work has already been documented extensively, so we focus on some of the non-technical aspects not covered in prior publications (Section 2). We then discuss our experience with industrial deployments, relating some factors that contributed to our successes (and failures) (Section 3). We then reflect on challenges (Section 4) and conclude with some thoughts about directions for the future (Section 5). We summarize a few main takeaways, grouped into a few themes:

Targets for Program Proof:

- —Cryptography and security protocols are a great domain for software proofs, since they are at the crux of the security of many systems, and have clear end-to-end security and correctness specifications. However, software proofs need not cover end-to-end system guarantees to be of value; e.g., systematically eliminating classes of vulnerabilities in specific software components is also broadly appreciated (Section 3).
- —Project Everest targeted standards-based software, anticipating that existing implementations could be replaced by verified, standards-compliant alternatives. However, we also worked on replacing implementations of non-standard software (e.g., proprietary components of the Windows kernel). In such cases, the effort spent on discovering the specification of an existing software component can be substantial, and tools to assist with specification discovery and validation (e.g., differential testing) are valuable (Section 3.3).

Factors for Industrial Deployments:

- —There is a widespread awareness of the value of software proofs, particularly in parts of the industry that build mission-critical software. Proof-backed software is a strong differentiator

[8] https://www.usenix.org/blog/usenix-announces-winners-2023-internet-defense-prize.
[9] https://github.com/cryspen/bertie.

and allows a small team of experts to deliver a component at a quality that would traditionally require a much larger team.
—A team embedded in an industrial-research setting helps with identifying high-impact targets and finding pathways to deployment. However, what is perhaps more important is for a research team to have a strong commitment to seeing the work through to deployment, to be willing to work closely with engineering teams and to be responsive to their needs, and to have the organizational support to make multi-year, risky bets.
—Formally verifying industrial-grade software at scale is feasible, though it still requires considerable effort. It is possible for engineers and proof experts to develop a large body of formally verified code (at the scale of a million lines of software) and maintain it over several years.
—Deployments of verified software require careful tradeoffs between formal guarantees, practical utility, and ergonomic usage. While minimizing the **trusted computing base (TCB)** is important, it is not always the leading factor in enabling adoption (Section 4.2).
—Developing components that can be integrated piecewise is a successful path towards deployment. Producing large monolithic systems requires too much integration work and ultimately might still not fulfill all the requirements of the systems they are trying to replace (Section 3.2).

Proof Engineering:

—Research-grade proof tools need a significant engineering investment to enable productivity at scale. Many of the engineering challenges of proof development and maintenance are similar to regular software engineering. However, proof engineering itself presents unique challenges, and balancing tradeoffs between proof automation and predictability remains an important research problem (Section 2.1.8).
—Employing higher-order coding, specification, and proof styles can simplify proofs, promote abstraction and reuse, and when used in combination with metaprogramming can yield large amounts of efficient verified code at a lower proof overhead, and with zero-cost abstractions. However, devising such proof styles requires a high level of expertise and fluency with advanced proof techniques (Section 2.1.7).

Future Directions:

—Although Project Everest focused specifically on co-developing a new language for program proofs with its applications, future projects might consider aiming to integrate proof technology within existing languages, potentially broadening the reach of the proof techniques. That said, the overhead of authoring proofs remains high and optimizing the development experience for proof-authoring over code-authoring might continue to make sense. Hybrid approaches that use distinct tools for code authoring and proof development may also be viable (Section 5.1).
—Two emerging trends augur well for reducing the cost of formal proofs. First, advances in programming languages, program logics, and proof tools continue to lower the cost of program proof. Notably, Rust as a systems language offers strong built-in memory safety guarantees, eliminating some proof obligations. Second, the emergence of generative AI models has triggered a burst of interest in using AI to automate a variety of specification, proof authoring, and maintenance tasks, aiming to lower the expertise needed to use proof-oriented languages (Section 5.3).

A Note on the Authors. This document describes the work and insights of a large team of people who contributed to Project Everest over the course of several years. A full list of contributors to Project Everest is in Appendix A.

2 Core Research Contributions

In this section, we present a brief overview of the main research contributions of Project Everest. These results were published in dozens of peer-reviewed papers, though we provide here a bird's eye view, focusing on the main themes of our work rather than on specific results, in four main areas: program proof tools, verified cryptography, parsers and serializers, and protocols.

2.1 Program Proof Tools

A basic tenet of our methodology was to build verified software from scratch, rather than attempting to verify existing implementations developed in other, general-purpose languages. We believed strongly, both then and now, that structuring code with proofs in mind would make the verification process more tractable. This view was also informed by experience in the broader program verification community, where projects had generally succeeded when building verified software from scratch [126, 142]. We also strongly believed in the promise of using SMT solvers (notably Z3 [71]) integrated with proof-oriented programming languages, to build verified software at scale.

2.1.1 Choosing a Language. An initial challenge was to consolidate on a single language. The project was initially a mixture of Dafny and F⋆, reflecting our backgrounds. In fact, the first version of Vale, our assembly language verifier (Section 2.1.6), was built using Dafny [46], and a later version was built using F⋆ [93]. As one can imagine, there were strong incentives to consolidate on a single tool, for many reasons. Maintaining and improving two tools was too costly, and we wanted to provide a single theorem to cover all of our verified code, which would have been (and still is!) hard to achieve when using multiple proof frameworks. There was much debate about the pros and cons of each language, some of which we summarize below.

In Dafny's favor:

- Dafny was more mature in 2016, had a better **interactive development environment (IDE)**, and better error messages when SMT proofs failed.
- Dafny had already been used to build complex verified systems like Ironclad (which included verified cryptographic primitives) and IronFleet, whereas the existing experience with F⋆ was more on applications related to compilers and programming languages.
- Dafny had built-in support for imperative programming and its syntax was more familiar to systems programmers.

In F⋆'s favor:

- By 2016, F⋆ had acquired a C backend through Low⋆ and KaRaMeL, while Dafny focused on producing C# code.
- F⋆ was more expressive than Dafny, supporting higher-order programming as well as higher-order proofs.
- F⋆ could model deallocation of memory, and distinguish stack and heap allocated memory, whereas Dafny assumes the use of automated memory management.
- F⋆ supported type abstraction and modularity, important for the style of cryptographic proofs we aimed to do. Dafny's module system was less mature and it still lacks other abstraction mechanisms like higher rank types and typeclasses.
- miTLS, a centerpiece of the project, was developed in F# and F7, with parts of it already ported to F⋆.

—F*'s dependent type checker already provided support for proofs by normalization, higher-order unification, and symbolic execution—it later evolved to also support tactic-based proofs. As such, it was less dependent on SMT-only proofs.
—Perhaps most significantly, the F* language developers were closely involved in the project and willing to evolve F* to meet the goals of Project Everest.

It is also worth mentioning that the Lean proof assistant [70] was being developed at MSR at the same time. However, Project Everest was heavily oriented towards using SMT solvers for proof automation, rather than interactive proof in a system like Lean. We did aim to use Lean as an interactive proving backend for some kinds of proof obligations produced by F*, but despite some work in this direction, the difficulties of integrating Lean with F*, given subtle logical differences (e.g., Lean is intensional whereas F* is extensional), were too great.

After considering all of these arguments and much debate, we eventually consolidated around F*, though using a young language at a large scale was also a major source of risk, and improving the language consumed a significant portion of the project's resources. Nevertheless, we also saw it as an opportunity to drive F*'s development.

*Evolving F**. Originally designed and implemented in 2011 [198], F* built on two prior languages from MSR, F7 [24] and Fine [197]. F* circa 2011 was notable for its combination of affine and (value-)dependent types. By 2015, we had found that this system was insufficiently expressive for the kinds of proofs we wanted to do, and that the affine types added a lot of complexity to F*'s metatheory. As such, in 2015, the language was completely redesigned and re-implemented, jettisoning affine types,[10] and generalizing it to support full dependent types, rather than just value dependency [200].

In terms of language features and verification methodology, a few main themes stand out: The development of *effect systems* in F*; embedded DSLs for low-level code, notably Low* and Vale; extensive use of higher-order programming and metaprogramming, combined with compile-time specialization; and the emergence of a proof style that used SMT solving heavily, but with the strategic use of proofs by normalization, reflection, and tactics for better proof performance and predictability.

2.1.2 Effect Systems in F.* A core organizing principle of F* is its integration of an *effect system* to enable imperative programming with side effects to be smoothly integrated within a dependently typed logic. The essential idea is to represent programs in an implicitly monadic form that enables a generic form of **verification condition (VC)** generation, through the use of *Dijkstra monads* [200, 203]. Dijkstra monads were theoretically investigated and further developed in a sequence of follow-up works [6, 147, 218]. Others have continued to study Dijkstra monads in a variety of settings, for infinite program executions [13, 191], for secure compilation [11, 12], and other applications.

For Project Everest, we primarily used Dijkstra monads to model effects of mutable state and non-termination, specifically in the definition of Low*, described next. Later, we also used it in the definition of a metaprogramming language, as described in Section 2.1.7.

Additionally, we developed a methodology for proving relations among monadic programs, particularly probabilistic equivalences [100]. At one stage, we had hoped to use this framework to conduct cryptographic proofs based on probabilistic equivalences; however, it was technically very difficult for our relational proofs to scale to handle all of TLS. Nevertheless, others used our monadic framework for relational verification to analyze TLS extensions (such as Encrypted Client Hello

[10]With the benefit of hindsight, given the remarkable trajectory of Rust and its use of affine types, one might argue that F* should have retained support for affine types, allowing it to more easily verify Rust-style code today.

[183]) to prove various indistinguishability-based privacy properties [144]. Later we generalized the idea behind Dijkstra monads to relational verification [148], which served as the foundation of SSProve [3, 107], a tool following the SSPs methodology and providing an embedded language for machine-checked cryptographic proofs in Rocq.[11]

Further innovations on F⋆'s effect system have generalized it to support arbitrary indexed effects, beyond Dijkstra monads [179]. F⋆'s effect system has also been used in secure compilation—an active area of ongoing research—enabling verified code to securely interoperate with unverified contexts [11, 12].

2.1.3 Low⋆: A Shallow Embedding of a C-Like Language in F⋆. Low⋆ [173] is a subset of F⋆ modeled after a fragment of C, with support for mutable state, pointers, arrays, stack and heap allocation, and machine integers. It is designed primarily to support proofs of array-manipulating sequential C programs, such as those used in cryptographic routines. Low⋆ grew from previous prototypes which had aimed for a more general language design, aiming to support a mixture of garbage-collected and explicitly managed memory, inspired in part by Cyclone [120, 199], but built on top of the OCaml runtime. This proved to be too complex, and we eventually preferred a design that is closer to C, with explicit memory management.

The core program logic of Low⋆ is a Hoare-style logic with *dynamic frames* [124], similar in spirit to Dafny. In contrast to Dafny, Low⋆'s theory of memory locations includes support for assertions about location disjointness and inclusion, and it distinguishes stack and heap locations divided into user-defined regions; further, all of these features are developed as verified libraries rather than as built-in concepts. Using libraries allowed us to gain trust in our memory models, and perhaps more importantly, allowed us to develop several iterations of Low⋆'s proof-oriented libraries to enable expressing footprints of heap data structures abstractly, and to try different strategies for proof automation. An important aspect of Low⋆ is its incorporation of a logic of monotonic state [5], which we used to model features such as the idealized state of a distributed system, e.g., global logs of all messages exchanged in a protocol. For all of this, the expressive power of F⋆'s higher-order, dependently typed logic was crucial.

Low⋆ was used extensively in Project Everest—HACL⋆, EverCrypt, EverParse, and miTLS were all developed in Low⋆. The general proof style involved proving the code in Low⋆ correct against a *low-level* functional specification; followed by a proof in F⋆ relating the low-level functional specification to a more abstract high-level functional specification. This two-step process allowed factoring the proof effort, enabling different people to work on the two parts of the proof in parallel, and for a separation of concerns between reasoning about memory footprints and the absence of undefined behaviors from the high-level functional intent.

Shallowly embedding Low⋆ in F⋆ enabled Low⋆ code to be metaprogrammed, or assembled from higher-order combinators, promoting proof reuse and genericity. Figure 3 shows a small but representative piece of Low⋆ code from HACL⋆, making use of higher order programming and compile-time specialization to produce many verified implementations of a **key derivation function (KDF)** from a single generic implementation, with Figure 4 showing the corresponding C code generated from the verified Low* source code.

Ultimately, using a variety of proof styles, more than 100,000 lines of code were written and verified in Low⋆ by hand, and still more code was generated by metaprogramming.

Others used Low⋆ too, building verified garbage collectors for OCaml [190], verified measured boot firmware [204], and even as the basis of other DSLs such as for verifying reactive systems [187].

[11] Rocq is the new name of the Coq proof assistant.

```
let expand_st (a: fixed_len_alg) =
  prk    : B.buffer uint8 →
  salt   : B.buffer uint8 →
  saltlen : pub_uint32 →
  ikm    : B.buffer uint8 →
  ikmlen  : pub_uint32 →
  Stack unit
  (requires λ h0 →
    B.live h0 prk ∧ B.live h0 salt ∧ B.live h0 ikm ∧ B.disjoint salt prk ∧
    B.disjoint ikm prk ∧ B.length prk == hash_length a ∧
    v saltlen == B.length salt ∧ v ikmlen == B.length ikm ∧
    Spec.Agile.HMAC.keysized a (v saltlen) ∧
    B.length ikm + block_length a < pow2 32)
  (ensures λ h0 _ h1 → key_and_data_fits a;
    B.modifies (B.loc_buffer prk) h0 h1 ∧
    B.as_seq h1 prk == extract a (B.as_seq h0 salt) (B.as_seq h0 ikm))

let compute_st (a: fixed_len_alg) =
  tag: B.buffer uint8 {B.length tag == hash_length a} →
  key: B.buffer uint8{ keysized a (B.length key) ∧ B.disjoint key tag } →
  keylen: UInt32.t{ UInt32.v keylen = B.length key } →
  data: B.buffer uint8{ B.length data + block_length a < pow2 32 } →
  datalen: UInt32.t{ UInt32.v datalen = B.length data } →
  Stack unit
  (requires λ h0 →
    B.live h0 tag ∧ B.live h0 key ∧ B.live h0 data)
  (ensures λ h0 _ h1 → key_and_data_fits a;
    B.modifies (loc_buffer tag) h0 h1 ∧
    B.as_seq h1 tag == hmac a (B.as_seq h0 key) (B.as_seq h0 data))
```

```
#push-options "-z3rlimit_300"
inline_for_extraction noextract
let mk_expand
   (a:fixed_len alg)
   (hmac:Hacl.HMAC.compute_st a)
: expand_st a = λ okm prk prklen info infolen len →
  let tlen = Hash.Definitions.hash_len a in
  let n = len /. tlen in
  Math.Lemmas.lemma_div_mod . . .
  let output = B.sub okm 0ul (n *! tlen) in
  push_frame (); (* lifetime of allocs that follow *)
  let text = B.create (tlen +! infolen +! 1ul) (u8 0) in . . .
  let tag = B.sub text 0ul tlen in
  let ctr = B.sub text (tlen +! infolen) 1ul in
  B.copy (B.sub text tlen infolen) info;
  . . .
  let h0 = ST.get () in (* ghost op to remember initial state *)
  B.fill_blocks h0 tlen n output a_spec refl footprint spec
  (λ i → (* loop body *)
    ctr.(0ul) <- Lib.IntTypes.cast U8 PUB (i +! 1ul);
    if i = 0ul then ( Seq.eq_intro . . .;
      hmac tag prk prklen text0 (infolen +! 1ul)
    ) else ( Seq.eq_intro . . .;
      hmac tag prk prklen text (tlen +! infolen +! 1ul)
    );
    Seq.unfold_generate_blocks . . .;
    B.copy (B.sub output (i *! tlen) tlen) tag
  );
  . . .
  if n *! tlen <. len then . . .;
  pop_frame() (* pop stack frame; end lifetime *)
#pop-options
```

```
(* Instantiations *)
let expand_sha2_256: expand_st SHA2_256 = mk_expand SHA2_256 Hacl.HMAC.compute_sha2_256
let expand_blake2b256: expand_st Blake2B = mk_expand Blake2B Hacl.HMAC.Blake2b256.compute_blake2b256
```

Fig. 3. A Low★ specification (left), generic implementation (right), and instantiations (bottom) of HKDF expand, a key derivation algorithm. The specification expand_st a, parametric in a fixed-length hashing algorithm, is the type of a stateful function that only allocates on the stack (enforced by the Stack effect label), describes anti-aliasing requirements (B.disjoint) and memory modification footprint (**modifies**), in the dynamic frames style. The last conjunct of the **ensures** clause is the functional correctness specification, describing the final value of the prk buffer. The type compute_st a is similar, and is the specification of an HMAC function. The implementation at the left is shallowly embedded imperative code, generic in an HMAC function for algorithm a. The code is proven in an intrinsic style, with lemma invocations interspersed with computational code. Note also the use of loop combinators (B.fill_blocks) provided by a library of utilities that package various common loop patterns and their invariants. The full implementation and proof is about 100 lines long. The #push-options directive instructs F★ to invoke Z3 with a higher resource limit—the default resource limit is just 5, indicating that this particular piece of code takes a significant amount of proof search to verify. Building specialized implementations, e.g., expand_sha2_ 256 is just a one-line instantiation of the generic mk_expand. The qualifiers **inline_for_extraction noextract** indicate that mk_expand is not to be extracted itself, but is to be inlined in its callers, and specialized at the call site by F★ before extraction to C via KaRaMeL. This allows one to easily obtain specialized, efficient implementations for many algorithms from a single HKDF implementation.

```
/**
Expand pseudorandom key to desired length.

@param okm Pointer to `len` bytes of memory where output keying material is written to.
@param prk Pointer to at least `HashLen` bytes of memory where pseudorandom key is read from.
           Usually, this points to the output from the extract step.
@param prklen Length of pseudorandom key.
@param info Pointer to `infolen` bytes of memory where context and application specific information is read from.
            Can be a zero-length string.
@param infolen Length of context and application specific information.
@param len Length of output keying material.
*/
void Hacl_HKDF_Blake2b_256_expand_blake2b_256(
  uint8_t *okm, uint8_t *prk, uint32_t prklen,
  uint8_t *info, uint32_t infolen, uint32_t len
)
{
  uint32_t tlen = 64U;
  uint32_t n = len / tlen;
  uint8_t *output = okm;
  KRML_CHECK_SIZE(sizeof (uint8_t), tlen + infolen + 1U); //Check that allocated size does not exceed SIZE_MAX
  uint8_t text[tlen + infolen + 1U];
  memset(text, 0U, (tlen + infolen + 1U) * sizeof (uint8_t));
  uint8_t *text0 = text + tlen;
  uint8_t *tag = text;
  uint8_t *ctr = text + tlen + infolen;
  memcpy(text + tlen, info, infolen * sizeof (uint8_t));
  for (uint32_t i = 0U; i < n; i++)
  {
    ctr[0U] = (uint8_t)(i + 1U);
    if (i == 0U)
    {
      Hacl_HMAC_Blake2b_256_compute_blake2b_256(tag, prk, prklen, text0, infolen + 1U);
    }
    else
    {
      Hacl_HMAC_Blake2b_256_compute_blake2b_256(tag, prk, prklen, text, tlen + infolen + 1U);
    }
    memcpy(output + i * tlen, tag, tlen * sizeof (uint8_t));
  }
  if (n * tlen < len)
  {
    ctr[0U] = (uint8_t)(n + 1U);
    if (n == 0U)
    {
      Hacl_HMAC_Blake2b_256_compute_blake2b_256(tag, prk, prklen, text0, infolen + 1U);
    }
    else
    {
      Hacl_HMAC_Blake2b_256_compute_blake2b_256(tag, prk, prklen, text, tlen + infolen + 1U);
    }
    uint8_t *block = okm + n * tlen;
    memcpy(block, tag, (len - n * tlen) * sizeof (uint8_t));
  }
}
```

Fig. 4. C code generated by KaRaMeL for HKDF expand instantiated with Blake2b-256, corresponding to the Low⋆ code in Figure 3. KaRaMeL propagates stylized comments in the F⋆ source code to the generated C code. It can also be instructed to add some defensive checks like KRML_CHECK_SIZE to ensure that allocations do not exceed SIZE_MAX, a platform-specific bound not enforced by Low⋆ as used in HACL⋆—more recent uses of Low⋆, Steel, and Pulse include a more precise, platform-neutral model of C's size_t type.

2.1.4 Revisiting Low⋆ with Separation Logic & Concurrency. With experience, we identified patterns of usage for Low⋆ that made it possible to develop the integer-array-processing code typical in HACL⋆. However, after completing a few significant proofs in Low⋆, including as early as a 2017 proof of the TLS record layer [72], we had also come to realize that reasoning about memory footprints in Low⋆ was difficult and scaled poorly to data structures with many pointers.

Despite a significant revision of the Low⋆ libraries, which improved the situation somewhat, the core of the problem was that Low⋆ proofs of heap properties are entirely dependent on SMT solving, and we found that proofs of programs that involved more than a few disjoint mutable memory locations, or program fragments that involved longer chains of mutations often could not be verified automatically in a reasonable amount of time, and required the user to supply additional lemmas and assertions. This is particularly pronounced when verifying pointer data structures: since Low⋆'s logic lacked any structural notion of separation, proofs of such programs, though possible, can be tedious and slow.

For instance, a doubly linked-list library took one team member more than 6 months to complete, requiring many layers of abstractions to tame reasoning about memory footprints. As another example, HACL⋆'s "streaming" APIs required simultaneously reasoning about modifying heap state, while allocating temporaries on the stack and juggling multiple pieces of data (key state, block algorithm state, user data, output array). This piece of code was written in a very manual style, performing memory reasoning by explicitly calling lemmas, disabling non-linear arithmetic, hiding specific definitions and lemmas from the solver, and splitting large stateful functions into smaller pieces that required writing and maintaining complex descriptions of intermediary states. Problems with scaling automated memory reasoning in Low⋆ has also been observed by other authors [136].

Further, Low⋆ has no support for concurrency. Our plan for reasoning about the concurrent uses of the top-level TLS interface of miTLS-fstar involved delegating the management of disjoint connections to unverified client code.

Recognizing these limitations, in 2018 we began to investigate the use of CSL [160, 184] to structure proofs of memory footprints. By then, Iris [121] had emerged as a powerful, unifying foundation for a modern CSL, though focused on interactive proofs in Rocq. We aimed to adapt some of its ideas for use in a setting with dependent types and SMT solving.

Our first attempt involved building a separation-logic-based memory model and VC generator using Dijkstra monads, in a style similar to Low*. Relying on Meta-F⋆ [149], a metaprogramming framework described more in Section 2.1.7, we also developed tactics to automate a class of proofs in separation logic. However, our model was too simplistic and did not support all the usual separation logic connectives, and lacked expressive power. Additionally, although our tactic-based proofs did not suffer from the unpredictability of SMT-only reasoning, proofs using this approach were considerably slower than even Low⋆, even for small programs.

We then developed SteelCore [202], encoding an Iris-inspired CSL in F⋆ by building on our prior work on monotonic state [5]. In 2020, we built Steel [95], libraries that, like Low⋆, shallowly embedded a C-like language in F⋆, but this time with a concurrency model and tactics to reason about heap footprints and framing expressed in separation logic. The result is an SMT-assisted program verification tool where heap reasoning is structured through a dedicated separation logic tactic rather than by the SMT solver.

Our goal was for Steel to replace Low⋆. However, Low⋆ continued to be heavily used in other parts of Project Everest while Steel was being developed, and ultimately, porting all our Low⋆ code to Steel was too labor intensive to attempt seriously. Besides, by 2021, the Project Everest team had begun to move on to other topics.

As Project Everest wound down, Steel was used to build other significant verified systems, including a high-performance concurrent state machine monitor [16] and a security hardened memory allocator [180]. Steel itself evolved with a new, more foundational and more expressive logical model into Pulse [80], which is actively developed today.

Low⋆ remains the implementation language of HACL⋆. The experience is one illustration of the challenge of co-evolving a language and its applications—changing tires on a moving car is not easy!

2.1.5 KaRaMeL. Accompanying Low⋆, and later Steel and Pulse, is KaRaMeL,[12] a compiler that translates OCaml-like programs produced by F⋆'s extraction pipeline (F⋆'s extraction is modeled after Rocq's extraction to OCaml [143]) to C code. KaRaMeL is designed to produce *readable* C code, aiming to enable adopters of our code to consume it simply as source code and potentially even maintain it (without proofs) in our absence. KaRaMeL, and the reassurance it provided to consumers of our code of being able to read and maintain the generated C code in an emergency, was a significant enabler of Project Everest's successful code deployments—more on this in Section 3.

The Input Language of KaRaMeL. In the 2017 paper that introduces Low⋆ [173], we formalized the input language to KaRaMeL as a minimalist first-order fragment of C, and proved on paper the correctness of a translation from this language to another, lower-level language modeled after an internal language of the CompCert [142] C compiler. This provided us with some confidence that the design of our approach was on a sound basis, though the implementation of the KaRaMeL compiler was in unverified OCaml.

Additionally, with time, KaRaMeL grew to handle many more features than our first formal model, though a guiding principle for KaRaMeL was (and remains) that features should be supported only insofar as they admit a predictable translation to C with no overhead. Notable features include support for data types and pattern-matching; whole-program monomorphization to compile prenex polymorphism down to C; a small configuration language for "bundles," enabling recombining F⋆ modules into C translation units, eliminating unused declarations, and establishing public APIs; various analyses to eliminate unused function arguments, unused local variables, unused type arguments, and unused type fields; some "peephole" optimizations, e.g., to rewrite the ML-style whole-record-update `r := {!r with f = v}`, into a C field-update `r->f = v`; and a litany of small cosmetic optimizations to recover good code quality after F⋆'s monadic encoding, all developed in response to specific requests by KaRaMeL's code consumers.

Nevertheless, KaRaMeL cannot be used to translate all well-typed Low⋆ programs to C. Instead, it handles Low⋆ programs that after compile-time specialization and extraction to OCaml do not use closures, make use of library operations to explicitly manage memory (including a distinction between stack and heap), and are well-typed in OCaml's typing discipline (without the use of escape hatches like `Obj.magic`). As such, the process of developing Low⋆ code also involves being mindful of KaRaMeL's restrictions, iterating with F⋆'s compile-time specialization and metaprogramming features to ensure that after extraction, the code is KaRaMeL-compatible.

Architecture of KaRaMeL. KaRaMeL was designed around a "nanopass" architecture. Each cosmetic optimization, no matter how small, is conceptually expressed as a single nanopass, relying on auto-generated visitors [170] to reduce implementation burden and to facilitate audit, as KaRaMeL is part of our TCB.

For instance, the compilation of data types is expressed as a sequence of schemes. First, data types with one constructor taking a single argument are eliminated entirely. Next, data types with n constructors each with zero arguments are compiled to C enums. Next, data types with a

[12] KaRaMeL was initially named KReMLin, for "K&R meets ML," but was renamed in 2022.

single constructor are compiled to a C struct. Next, data types with n constructors, only one of which has arguments, are compiled to a C struct with a tag, directly followed by the non-constant constructor's arguments (which may be possibly uninitialized). Finally, remaining data types are compiled to a tagged union scheme in C.

Such complexity is necessary for code quality, but also simply for getting C code to compile. Without the described optimizations for representing datatypes, the Microsoft Visual C compiler refused to compile our code for miTLS because it exceeded a hard-coded limit for the nesting depth of C structs. Thankfully, the nanopass architecture, combined with a widespread use of automatically generated visitors, makes this complexity tractable. We currently have more than 80 nanopasses in KaRaMeL. Despite the many passes, KaRaMeL execution time is usually not the bottleneck in a build, since verification time in F⋆ and Z3 usually dominates.

KaRaMeL continues to evolve and now supports extraction to Rust from Low⋆ and other F⋆ DSLs.

2.1.6 Vale: Verified Assembly Language for Everest. While Low⋆ was our primary language for proofs of C code, we knew from our experience with Ironclad [109] that to achieve performance competitive with existing commercial cryptographic libraries, we would need a way to write and verify assembly code. Cryptographic libraries, like OpenSSL, use aggressive tricks to optimize their assembly code [46], not just to use platform-specific hardware instructions (like AESNI [103] or NEON [17]), but also to carefully manage data movement between registers and memory, to unroll loops, and to interleave memory fetches with computation so as to keep the CPU pipeline saturated. To produce code like this, OpenSSL uses a mix of Perl, C preprocessor macros, and hand-written assembly. We wanted a tool that could replicate this flexibility while providing formal guarantees about the code, ideally without adding the tool itself to the TCB. Since we anticipated targeting multiple hardware platforms, we also wanted to keep the tool agnostic about the hardware.

Architecture-Specific Deep Embeddings in Dafny and F⋆. Vale (Verified Assembly Language for Everest) is a language with associated tools [46]. The language looks like an assembly language augmented with standard verification features, including methods with pre-/post-conditions and modifies clauses, and the ability to define ghost state, assert properties, and call lemmas. One design choice that worked well for our use case was the decision to only support structured control flow. This kept both the code and the reasoning about the code cleaner and simpler, and in practice, it did not prove unduly restrictive when writing cryptographic routines, since existing cryptographic code bases also tend to follow this practice.

To support reasoning about Vale code, we developed a (trusted) deep embedding of each target hardware platform in a verification language (as we discuss below, initially we used Dafny, but later we shifted to F⋆). The semantics define the state of the hardware (e.g., the registers, various CPU flags, and byte-addressable memory) and a relation specifying how the state could evolve based on executing code written with the subset of instructions we anticipated using in our cryptographic code. To make verification more efficient, we also defined a series of Hoare-logic wrappers around each instruction, in order to hide the full complexity of the semantics.

Given an assembly program written in Vale, the Vale tool compiles it into code in the target verification language. Specifically, Vale (a) creates executable code in the verification language that constructs an AST representing the assembly code, and (b) provides a series of lemmas that walk the verification language's automation through the effects of executing the AST. Finally, a simple trusted printer written in the verification language emits the AST as a series of vanilla assembly instructions. The core of Vale is agnostic about which verification language it targets, but it includes a prover-specific backend for emitting properly formatted code.

Initially, Vale did not perform type-checking at the Vale-source level; instead it simply relied on the underlying verification language to catch and report typing errors. Eventually, however, we added a proper type checker to Vale itself. This improves error reporting, but more importantly, it enables Vale to emit more efficient code for verification purposes. In particular, Vale includes a custom range analysis for integer values bounded by constants, which occur ubiquitously when reasoning about assembly code. Leveraging this domain-specific property enables Vale to perform better analysis than the generic analysis performed by either Dafny or F*, and hence produce more efficient code for reasoning about the assembly.

In general, the Vale design worked well for us. Vale's flexibility enabled us to produce assembly code for many different ISAs, from various flavors of ARM, x86, and x64 for Everest purposes, to more exotic hardware (e.g., a custom 256-bit big number accelerator and a tiny 16-bit TI MSP430) in later projects [228]. We gained this flexibility *without* adding Vale to our TCB; any mistakes in the tool result in verification failures, not unsoundness.

Vale also includes enough flexibility to produce state-of-the-art performance (see Section 2.2 for performance results). For example, the Vale language supports inline procedures, conditionals, and loops, which are compiled to executable code-producing functions in the target verification language. Hence, a Vale developer can write and verify an inline loop at the Vale source level that is dynamically unrolled during code production. Vale procedures can also take generic operand arguments, which allows the developer to match, for example, the tricks that OpenSSL uses to minimize data movement between registers when computing the SHA-256 hash function. In short, Vale allows developers to conveniently write and verify the many different ingenious tricks they devise to eke out more performance.

Our deep embedding approach proved useful in multiple dimensions. For instance, it made it trivial to produce AST printers for different assemblers, including the GNU assembler, the MASM assembler, and `gcc` inline assembly. It also allowed us to develop verified information flow analyses to rule out classes of side-channel attacks, which we describe next.

A Certified Taint Analysis for Tracking Side-Channels. We invested considerable effort in proving the absence of side channels in our cryptographic code, a crucial property often neglected in other cryptographic verification efforts [15, 27, 63, 85, 169, 206, 208, 224]. A side channel leaks secret information implicitly, e.g., based on the time it takes the code to execute or which memory addresses are (or are not) in the cache after the code executes. Side channel attacks have been used to devastating effect on deployed cryptographic code [4, 14, 30, 50, 88, 129, 167, 223]. Rather than directly prove side-channel freedom for each piece of assembly code, we augmented our ISA semantics with an adversarial trace of operations representing information (like branches taken or memory addresses accessed) that might leak via side channels. We then wrote a conservative, executable taint tracker in the target verification language, and proved (as a one-time effort) the tracker sound against our augmented ISA semantics; in other words, if the taint tracker approved a given AST, then the execution of that AST would achieve non-interference with respect to cryptographic secrets.

Taint tracking in the presence of pointers is very difficult, in both theory and practice. In our domain, however, because the developer is already proving the functional correctness of their code, we can make the problem quite tractable. In particular, we ask the developer to augment their load and store operations to indicate whether they are accessing secret or public data. The functional verification ensures these labels are accurate, allowing the taint tracker to assume they are correct, giving it perfectly precise taint tracking into and out of memory. This approach was effective in certifying all of our Vale code as free of basic digital side channels, and in the process, it uncovered a place where OpenSSL's code (which we had ported to Vale) left secret-tainted data in memory after it terminated [46].

Heaplets. We also encountered various limitations with Vale's design. For example, in our initial work with Vale, reasoning about memory usage proved to be quite painful. The semantics of, e.g., writing from a 64-bit register to memory dictated breaking up the 64-bit value into eight bytes via a series of divisions and remainders and then writing to eight consecutive locations in the byte-addressable machine memory (and similarly for reading a 64-bit value from memory). Composing multiple such operations produced a huge amount of mathematical reasoning as well as reasoning about the disjointness of the affected memory regions. We ultimately addressed this via a somewhat ad hoc "heaplet" abstraction over the raw memory. This abstraction divided the memory into disjoint heaps, each of which could view memory as a sequence of a different type (e.g., bytes, 64-bit words, or 256-bit words). A global invariant kept the heaplets in sync with the physical memory, and one-time proofs established the soundness of performing, say, 64-bit reads and writes on a corresponding 64-bit heaplet. Often we used a heaplet for each logical array that the code operated on to simplify reasoning about disjointness.

Multiple ISAs. While our deep embedding of machine semantics brought several advantages, it also added a significant amount of boilerplate to the process of defining a new ISA, or extending an existing ISA. Creating a new ISA entailed not just defining the semantics, but also creating (fairly boring) Hoare rules for each instruction, and then also defining and proving various abstractions about the ISA (e.g., the heaplets described above, or abstractions for the stack). This led to waning enthusiasm for expanding to new platforms. In post-Everest work, we developed techniques to mitigate some of this tedium [228], but room remains for additional automation.

Optimizing VCs with Proof-by-Reflection in F⋆. Perhaps the largest difficulty we encountered was that Vale's approach of embedding assembly reasoning into an existing verification language gave up control over how the verification of the code proceeded, resulting in excessively large, complex VCs, which Z3 (the underlying SMT solver used by both Dafny and F⋆) struggled to discharge. For example, suppose that given an assembly procedure that copies a small value in register `rax` into `rbx`, and then adds `rax` to `rbx` two times, we want to prove that the final value of `rbx` is three times the value in `rax`. Written as a program in either of the verification languages (e.g., using two variables to represent the registers), this produces a simple, easily discharged VC. However, when embedded via Vale, the VC becomes far more complex. Instead of encoding `rax` and `rbx` as distinct variables, they are instead indices into the machine's register file, so they come with side conditions that each index is in bounds and that a register is a valid operand for each place it's used in the assembly instructions. Reads and writes to the registers become map select/update operations, and the solver must reason about whether or not `rax` and `rbx` are disjoint. In long code blocks, the solver must repeatedly apply various select/update axioms to discover the current values of various pieces of state, and worse, most of this reasoning is discarded when the solver decides to backtrack. As a result, verification time explodes, making it tedious and inefficient to write and debug proofs about the code.

This poor verification performance, along with a desire for a unified verification language and security theorem (Section 2.1.1), led us to switch from using Dafny as a Vale backend to using F⋆. In F⋆, we developed a custom VC generator for Vale as a normal F⋆ function that F⋆ applies to our assembly code [93], in a style similar to proof-by-reflection. The VC-generator is executed using F⋆'s advanced normalization features to automatically simplify the VC into a form that captures the essence of the assembly code's mathematical reasoning and eliminates the overhead that previously arose from the deep embedding. F⋆ then sends the simplified VC to the SMT solver as normal. This approach resulted in significant performance improvements, particularly for larger assembly procedures where developers tend to spend most of their time. This work also inspired later work that makes it simpler and more efficient to embed DSLs in F⋆ [112, 201]. Figure 5 shows a verified implementation of Poly1305 in Vale.

```
procedure Poly1305(
        inline win:bool,
        ghost ctx_b:buffer64,
        ghost inp_b:buffer64,
        ghost len_in:nat64,
        ghost finish_in:nat64)
    {:public} /*visible from outside the module */
    {:quick}  /*enable F*'s normalization-based VC generator */
    {:exportSpecs} /*export specs for interop with Low* */
    lets
        ctx @= rdi; inp @= rsi; len @= rdx; finish @= rcx; ...
    reads
        heap0;
    modifies
        rax; rbx; rcx; rdx; rsi; rdi; rbp; rsp; r8; r9; r10; r11; r12; r13; r14; r15;
        efl; heap1; memLayout; stack; stackTaint;
    requires
        rsp == init_rsp(stack);
        is_initial_heap(memLayout, mem);
        buffers_disjoint(ctx_b, inp_b);
        validDstAddrs64(mem, ctx_in, ctx_b, 24, memLayout, Public);
        validSrcAddrs64(mem, inp_in, inp_b, readable_words(len_in), memLayout, Public); ...
    ensures
        modifies_buffer(ctx_b, old(mem), mem); ...
        finish_in == 0 ⟹ ...;
        finish_in == 1 ⟹ h10 == poly1305_hash_all(modp(h_in), key_r, key_s, inp_mem, len_in);
        rsp == old(rsp);
        win ⟹ rdi == old(rdi);
        win ⟹ rsi == old(rsi); ...
{
    CreateHeaplets(list(
        declare_buffer64(inp_b, 0, Public, Immutable),
        declare_buffer64(ctx_b, 1, Public, Mutable)));

    let key_r0 := buffer64_read(ctx_b, 3, heap1); ...
    Mov64(rax, ctx);
    Mov64(r11, inp);
    inline if (win)
    {
        Mov64(ctx, rcx); Mov64(inp, rdx); Mov64(len, r8); Mov64(finish, r9);
    }
    Store64_buffer(heap1, ctx, finish, 184, Public, ctx_b, 23);

    Push_Secret(h1); ... Push_Secret(r15);   // Save callee-saved registers
    let hprime := Poly1305_impl(key_r, key_s, ctx_b, inp_b, finish_in);
    Store64_buffer(heap1, ctx, h0,  0, Public, ctx_b, 0);
    Store64_buffer(heap1, ctx, h1,  8, Public, ctx_b, 1);
    Store64_buffer(heap1, ctx, h2, 16, Public, ctx_b, 2);
    Pop_Secret(r15);... Pop_Secret(h1);     // Restore callee-saved registers
    Mov64(ctx, rax);
    DestroyHeaplets();
}
```

Fig. 5. A verified implementation of Poly1305 in Vale (shown partially), porting an implementation from OpenSSL. Vale includes features similar to compile-time macros to allow the code to be specialized for different calling conventions, e.g., the `win` parameter indicates whether the Windows calling convention is used. The attribute `quick` enables Vale's normalization-based VC generator certified in F⋆. The specification includes a full functional correctness specification, relating the assembly code to `poly1305_hash_all`, a high-level functional specification in F⋆.

```
let dom = [
  TD_Buffer TUInt8 TUInt64 {modified=true;  strict_disjointness=false; taint=MS.Public};
  TD_Buffer TUInt8 TUInt64 {modified=false; strict_disjointness=false; taint=MS.Public};
  TD_UInt64;
  TD_UInt64
]
let lowstar_poly
: IX64.as_lowstar_sig_t_weak_stdcall code_poly dom [] _ _ (W.mk_prediction code_poly dom [] (poly_lemma code_poly IA.win))
= IX64.wrap_weak_stdcall _ _ _
```

Fig. 6. A Low⋆ wrapper for the Vale implementation of Poly1305 shown in Figure 5, produced using our generically verified interoperability wrapper. The dom list describes the types of the arguments to the Vale procedure, including two buffers and two 64-bit integers. The user-provided poly_lemma interprets the Vale specification in terms of F⋆'s state transformer monad, and is used by W.mk_prediction to prove that running the Vale code code_poly satisfies the Low⋆ signature, as_lowstar_sig_t_weak_stdcall.

Verified Interoperability between C and Assembly. Safe multi-language interoperability is a problem that has been studied extensively in the programming languages literature, particularly between high-level and assembly languages [66, 91, 115, 163, 164]. Much (though not all) of the prior work comes at the problem from the perspective of compiler correctness, type-safe linking between components written in two different languages, or even program equivalences that allow replacing a component written in a high-level language with optimized assembly code. In the context of Everest, our goal was to develop optimized cryptographic routines in both assembly and C, and to offer a unified cryptographic provider interface callable from C (i.e., EverCrypt) which could internally call either C or assembly implementations of cryptographic primitives, while providing a single *functional correctness* theorem. Our interoperation model was relatively simple, since although C programs could call into assembly routines, we did not need to support assembly calling back into C.

With Low⋆ and Vale both embedded in F⋆, we had an opportunity to provide a clean, integrated solution to this problem. As mentioned previously, Low⋆ is shallowly embedded in F⋆ as a state transformer monad, while Vale is deeply embedded in F⋆ with a semantics given in terms of a state-transforming interpreter of assembly instructions. Abstractly, to formalize a call from C to assembly, we reified the state of a Low⋆ program to call the Vale interpreter on a given piece of assembly code, while formalizing the calling convention in a platform-specific manner.

Initially, to support calling Vale procedures while passing a variable number of arguments, we wrote an ad hoc external tool to generate the necessary boilerplate code to build Low⋆ bindings to a Vale procedure [93]—this would then be checked for correctness by F⋆. Later, we developed a more sophisticated approach that used dependently typed arity-generic programming [9] to verify, once and for all, a Low⋆ wrapper for Vale procedures [172]. This approach involved a one-time effort to verify a generic and modular definition of interoperability (including calling conventions) that could then be easily extended to new platforms and new features. The code listing in Figure 6 shows a use of our generically verified interoperability wrapper producing a Low⋆ wrapper of the Vale implementation of Poly1305 shown in Figure 5.

We often employed these interoperability tools in our cryptographic libraries (Section 2.2) in order to replace fragments of C code with hand-optimized assembly, while providing a single correctness theorem.

Beyond Everest. While we designed Vale with cryptographic code in mind, subsequent work (by us and others) has used it in a variety of settings, including a software monitor for on-demand, user-mode, concurrent isolated execution [89], secure boot code burned into the mask ROM on

the OpenTitan security chip [228], an I/O separation kernel [225], and a verified WebAssembly sandboxing compiler [49].

2.1.7 Metaprogramming. A key aspect of our proof methodology was the use of metaprogramming to generate and verify code. We use the term to encompass two different styles.

Higher-Order Programming with Compile-Time Reduction. As in other dependently typed languages, the F⋆ compiler includes an abstract machine for reducing program fragments. This machine is capable of inlining, as is usual in many compilers, but during inlining it can also reduce lambda terms (including terms with free variables), simplifying branches and unrolling recursion. Using this facility, we programmed in a generic higher-order style and relied on the compiler to specialize the code to a closure-free form, enabling it to be compiled to efficient C code by KaRaMeL. This effectively allowed us to compile a much greater class of F⋆ programs to C, i.e., those that fit in KaRaMeL's input language *after compile-time reduction.* Three brief examples follow.

A first example is from EverParse [176], a verified parser generator. Rather than program a parser for each format by hand, we designed a higher-order library of parser combinators, proven correct compositionally. Then to parse a given format, we developed tools to generate the appropriate application of combinators, with F⋆ compile-time specializing the result into first-order code. We also made heavy use of partial evaluation, especially for 3D [201], to turn an interpreter for a DSL of parsers into a compiler, a technique known as a Futamura projection [96].

Another example is from HACL⋆: For cryptographic libraries, one wishes to provide specialized algorithms for cryptographic constructions that are parameterized by various algorithms. For example, an authenticated encryption construction could be instantiated with several different ciphers and message-authentication codes. Rather than program all the combinations separately, and do proofs for each of them, we programmed a single generic construction parameterized by the choice of primitives, proved it correct once, and then compile-time specialized it several times to obtained optimized implementations for each of the many choices of primitives. An example of this style was shown in Figure 3. In addition to code reuse, verifying generic combinators can sometimes be *easier* than verifying even a single specialized version, as the abstraction imposed by genericity can force a separation of concerns; e.g., the proof relies only on some generic algebraic property, rather than specific properties of the arithmetic operators.

In an extreme example of genericity, the Noise* project [112] developed a generic library of combinators for a family of key exchange protocols, doing proofs once that covered all 59 protocol variants in the family, in about 40,000 lines of generic, higher-order Low⋆ and F⋆ code. These variants could be specialized further to several hundred implementations, yielding more than 3 million lines of C code, if one were to generate them all.

Syntax Reflection. In 2017, inspired by work on Lean [70] which was being developed simultaneously at Microsoft Research, we developed general-purpose syntax reflection for F⋆ called Meta-F⋆ [149]. This is the basis of F⋆'s tactic engine, its typeclass system, and a general facility for metaprogramming by syntax reflection, allowing F⋆ meta-programs to inspect and generate F⋆ code. A unique aspect of F⋆'s metaprogramming system was that it was designed to be used in conjunction with VC generation and SMT solving.

While at the start of the project, we relied mostly on SMT solving for proofs, as Meta-F⋆ became available, we selectively moved to tactic-based proofs to automate certain classes of proof obligations, relieving the SMT solver of some of the burden of proving complex properties. For instance, many proofs involved reasoning about non-linear arithmetic, which is poorly automated by SMT solvers. However, many proofs are streamlined simply by rewriting terms into a normal form by associativity and commutativity—we relied on tactics for such proof steps. Similarly,

we developed tactics to convert reasoning about operations on bounded machine integers into properties of bit vectors, and then used Z3's bit vector solver.

Meta-F* was also a key enabler for our forays into separation logic. As explained in Section 2.1.4, Steel avoids using the SMT solver for heap reasoning. Instead, we developed tactics in Meta-F* for solving a small class of separation logic entailments, including finding frames for the application of the frame rule, introducing and eliminating existential predicates.

We also combined compile-time specialization with syntax reflection in HACL*, to implement more advanced forms of modular, compile-time specialization (resembling C++ template metaprogramming) [110].

2.1.8 Engineering. In addition to research advances around program logics and proof methodology, we invested significant effort in engineering aspects of the language and its ecosystem. We highlight a few significant elements here.

IDEs. Quick incremental feedback is extremely important when verifying a large program, and batch mode verification was too slow. We needed an IDE backed by a language server that enabled incremental verification. We started with the Atom editor, but by 2017, we had started to use fstar-mode.el, an Emacs-based interactive environment similar to IDEs for Rocq and other languages. This was the main IDE for F* and remains useful. A plugin for VS Code, fstar-vscode-assistant, was developed later and is used by many F* developers today.

Build System. We primarily used GNU Make—despite its many quirks, it was familiar to everyone. To verify Vale code (and to build Vale itself), we also used SCons, a Python-based build tool. To enable separate compilation, with incremental and parallel builds, we made many enhancements to F*.

CI. Perhaps the most important element in our engineering infrastructure was the CI system, enabling distributed collaboration at scale. We spent a huge effort on this, with at least one dedicated person maintaining the system for several years, through several iterations, with the system being overhauled completely at least 3 times. CI remains crucially important and continues to evolve, though the available tooling for this has also improved and become somewhat more standard, e.g., through the use of GitHub actions. A complete build of all the Project Everest code takes about 2.5 hours on our CI system today.

Project Everest was structured as a collection of several GitHub repositories, one for each sub-project. For instance, we had separate repositories for F*, KaRaMeL, Vale, HACL*, EverParse, miTLS-fstar, and so on. This reflected the relatively loose federation of the team spread across several organizations, in contrast to the monorepo style [171] that uses a single repository for multiple projects. This was in part because some of our repositories were pre-existing (e.g., for F* and Dafny, and these had uses independently of Project Everest), and in part because we wanted to minimize contention when pushing changes, making it easier for the various projects to experiment more freely.

Nevertheless, we also wanted to ensure that sub-projects remained working together as they evolved. Towards this end, we had a repository for synchronization among the projects which maintained a set of commit hashes for each sub-project recording the last known combination at which all the projects worked together, similar to git submodules but with a simpler workflow customized to our use case. To enable this style of development, we had a hierarchical CI system, with separate builds for each project, and then an Everest-wide CI system to build all of the main branches of the sub-projects together four times a day, using the latest versions available, and advancing the commit-hash-set on success. Build notifications were sent to a Slack channel.

This hierarchical approach had several benefits. It allowed each project to evolve separately, especially enabling development work to proceed in branches without need for global synchronization. Build times were also faster for individual projects, especially important in a setting with verification, since compilation time includes proof-checking and can be quite long. Additionally, the main branches of all the projects were synchronized every 6 hours, providing the usual benefits of CI: reacting quickly to broken builds; easily identifying breaking changes; with notifications to raise awareness of breaking changes.

We continue to maintain this hierarchical CI system today, though several other CI systems have also come up. For instance, F⋆ has a "check-world" build, which orchestrates around 21 build and test actions from a collection of "friend" projects as a workflow of GitHub actions. Separately, INRIA maintains a Nix-based CI system for HACL⋆ and related projects.

Setting Up a Consistent Development Environment. Owing to the hybrid Windows/Linux development environment and our tools' dependency on the OCaml ecosystem, new contributors found it hard to get a working development environment. We invested in a bash script (that grew to be several thousand lines long) to setup a fresh development environment while catching setup mistakes and/or known bugs (e.g., faulty versions of package dependencies). More recently, Nix has emerged as a more systematic way to address reproducible builds and dependencies.

2.1.9 Observations. We share some of our observations on our program proof methodology.

Our most significant decision was to co-develop program proof tools along with their applications, and to center our development around F⋆. Looking back, we believe we made the right choice. In 2016, the tools to build low-level verified software for the applications and scale we aimed for did not exist. Co-developing the toolchain and the applications allowed us to make progress and allowed us to succeed in most of the project's goals. It also yielded, in F⋆ and related libraries, a general-purpose program proof toolchain that continues to be used today for a variety of applications. Of course, we also encountered many challenges, which we discuss in Section 4.

Attempting a verification project today at a similar scale, we would consider F⋆ and newer frameworks built on top of it, notably Pulse, with KaRaMeL for extraction to C and Rust. However, as discussed in Section 5, one might also consider tools like Aeneas, Hax, Prusti, or Verus to verify Rust code directly with different tradeoffs, though none of these tools have yet been used to develop a body of verified software at the scale of what was developed in F⋆ by Project Everest. As such, we expect that any future large-scale verification effort would likely also evolve their verification tools and methodologies in service of their applications. Besides, the effort involved in developing provably correct software remains high and continued innovations in program proof tools and methodologies are crucial to making the discipline more cost effective.

2.2 Cryptographic Primitives and Constructions

Cryptographic primitives (e.g., for encryption or signatures) make attractive verification targets, since (a) they typically have a relatively succinct, mathematical definition of correctness; (b) they often lie on the critical path for application performance, motivating extensive and complex optimizations; (c) the amount of code needed for a given primitive is non-trivial but still within the reach of modern verification tools; and (d) tiny flaws in the implementation can deeply break the application's security. As a result, many lines of work [20] have tackled this problem. However, much of the work in this area has focused on portions of a cryptographic primitive (e.g., the field operations used within the routines for digital signatures), or ignored either performance or side channels (or both).

In contrast, Project Everest focused heavily on performance, since we expected few industrial users would be willing to use slower cryptography, even if it came with stronger guarantees. We

also focused on providing a comprehensive suite of fully verified primitives, so that applications (including TLS) would have everything they needed in a unified cryptographic provider. Finally, because side channel attacks have historically plagued real-world cryptographic deployments we took steps to ensure basic side-channel freedom in all of our verified cryptography.

In this section, we discuss our experience developing the HACL⋆ and ValeCrypt libraries of verified cryptographic primitives, and EverCrypt, a cryptographic provider.

2.2.1 Cross-Platform C Code With HACL⋆. Being implemented in Low⋆, HACL⋆ extracts to C code that can be compiled with widely used compilers such as GCC or clang. With the exception of **Single Instruction Multiple Data (SIMD)** optimizations, described later in this section, HACL⋆ implementations are intended to be portable, and therefore easily deployable on different platforms. At the time of writing, HACL⋆ provides the following functionalities: **authenticated encryption with additional data (AEAD)** (Chacha20-Poly1305), elliptic-curve Diffie-Hellman key exchanges (for both Curve25519 and Ed25519), signatures (Ed25519, ECDSA P-256, and RSA-PSS), hashes (the SHA2, SHA3, and Blake2 family of hashes, as well as the obsolete SHA1 and MD5), hash-based key derivation (HKDF, using either SHA2 or Blake2 as a basis), symmetric encryption (Chacha20), and message authentication codes (Poly1305, and the hash-based HMAC using any of the hashes provided by HACL⋆). The entire library consists of 212,000 lines of F⋆ code (specification and proof included), ultimately generating 101,000 lines of C code. In the rest of this section, we discuss several design choices and verification techniques used pervasively throughout HACL⋆.

Generic Integer and Array Libraries. Cryptographic implementations typically rely on the same set of core program constructs, namely, machine integers and arrays. To reduce code duplication and proof effort, we leveraged F⋆'s dependent type system to implement generic integer and array libraries, which provide both the core functionalities and many helpful lemmas and properties needed to implement a range of cryptographic primitives. Our dependently typed overloading is an alternative to typeclass-based overloading [212], allowing us to encode various subtleties, such as the partiality of operations to ensure the absence of overflow, and a distinction between public and secret data, as described next.

While Low⋆ provides builtin types and operators for machine integers (from 8 to 128 bits), directly using them can be tedious: for instance, performing an integer addition requires calling the addition function for a given integer type (e.g., uint8 or uint32), instead of using the standard + operator. To simplify the use of machine integers, we defined an abstract integer type on top of Low⋆'s machine integers, parameterized by a tag enumerating all known variants of integer types. We then redefined operators such as addition or multiplication to use the abstract integer type, and to be parametric in the integer width. Importantly, the tag is marked as an *implicit* argument: when writing cryptographic code, programmers do not need to explicitly specify it at each operation; F⋆ is in almost all cases able to infer it from the context.

In addition to abstracting over the integer width, the abstract integer type used in HACL⋆ is also parameterized by a *secrecy level*, distinguishing between public and secret data. This parameter allows restricting operations that can be performed on secret integers: any non-constant time operation (e.g., division), comparison (needed for branching), or array access can only be performed using public integers. As the integer type is abstract, users can only manipulate integers through operations exposed by the library, thus ensuring that cryptographic implementations satisfy the constant-time programming discipline, and are robust against basic digital side-channel attacks. Our formal model of Low⋆ [173] included a pen-and-paper proof of a secret-independence metatheorem proving the soundness of this type-abstraction-based approach to constant-time programming.

The array library offers an abstract, polymorphic type for Low⋆ arrays, as well as several generic, higher-order combinators for common operations, including iterating over the elements of an

array, or applying a fold operator for a user-provided function. This library also provides common lemmas and properties for these operators, for instance, that applying a fold operator with two equivalent functions yields the same result.

Importantly, both the generic integer and array libraries leverage F⋆'s metaprogramming facilities to offer *zero-cost abstractions*. Abstractions and combinators are inlined at compile-time to retrieve Low⋆ machine integers and C-like loops iterating over arrays respectively, without inducing any runtime cost. In particular, this prevents the need for function pointers when relying on higher-order combinators: functions passed as arguments are inlined in the body of the combinator.

SIMD Implementations. Many modern processors provide SIMD instructions, which allow performance optimizations based on vectorization, computing over 4, 8, or 16 integers in parallel. Cryptographic code is particularly amenable to SIMD-based optimizations, with several algorithms (including the Chacha20 stream cipher and the Blake2 hashes) deliberately designed to enable them. While platform-specific, SIMD instructions are typically available at the C level through the use of compiler intrinsics. Optimizing for a variety of platforms can however be tedious, requiring duplicating similar code, with minor differences to account for different platforms and levels of vectorization.

One key observation is that, despite small differences, cryptographic code relying on different levels of SIMD vectorization exhibits the same high-level code patterns, which only depend on the number of computations performed in parallel by SIMD instructions. Relying on F⋆'s support for dependent types, we developed cryptographic implementations generic in a vectorization level [168]. Similarly to our methodology for generic integers, we defined an abstract type for integer vectors, parameterized by the vector width. We then defined width-parametric operations for this datatype, such as vectorized arithmetic and bitwise operations, which abstract over the specifics of instructions on a given platform (e.g., ARM Neon vs. Intel AVX). Cryptographic implementations using this library could then be specialized using compile-time reduction, selecting the appropriate compiler intrinsics for a given platform and vectorization level.

This separation between a generic, verified implementation and its executable specializations had several benefits. First, it greatly reduced code duplication, as well as the time and effort needed to develop different vectorized implementations. Second, it also helped with the proofs, by abstracting over platform-specific details which are irrelevant to establish the correctness of implementations and also helping SMT-based proof search by removing irrelevant facts from the search space (Section 4.3). Third, relying on one generic implementation simplified maintenance, as well as extensions to novel platforms. When unstable proofs broke, only the generic implementation needed to be fixed. Further, adding support for, say, ARM Neon when Intel AVX was already supported, only required instantiating the abstract vector operations with the corresponding compiler intrinsics.

Of course, it was crucial to ensure that genericity and ease of verification was not done at the cost of performance. To reach this goal, F⋆'s support for dependent types and metaprogramming, allowing the use of zero-cost abstractions, was particularly significant.

Development of HACL⋆ continues today, to support new algorithms, new features as demanded by deployments, and for regular maintenance. One significant recent advance was the use of metaprogramming to support **Hybrid Public Key Encryption (HPKE)**, which we describe next.

HPKE. HPKE [22] is a recently standardized cryptographic construction used in several cryptographic protocols, including MLS [21] and TLS' Encrypted Client Hello [183]. The construction combines several cryptographic components: (a) a **key encapsulation mechanism (KEM)**; (b) a KDF; (c) AEAD. Additionally, the HPKE standard recommends multiple ciphersuites, allowing four different KEMs (P-256, P-521, Curve25519, Curve448), two KDFs (HKDF-SHA256, HKDF-SHA512), and three AEADs (AES-GCM-128, AES-GCM-256, Chacha20Poly1305), leading to 24 possible ciphersuites, and many more implementations to support platform-specific optimizations, such as

the SIMD vectorization previously described. We also implemented generic streaming APIs to transform block-based algorithms into a safe, high-level API, encapsulating a state machine over internal state—we describe this more in Section 4.3.

Given its structure, HPKE is particularly well-suited to HACL⋆'s generic verification methodology. Similarly to the SIMD implementations, we develop a generic, verified HPKE implementation, which is parameterized by the ciphersuite used, and thus by, e.g., abstract encryption and decryption functions. The compile-time specialization is however more complex: while inlining was sufficient for previous applications, it is here important for code quality to preserve the structure of the call-graph to, e.g., call into an AEAD library after specialization instead of entirely inlining it. To this end, we rely on F⋆ metaprogramming facilities to perform call-graph rewriting [110].

The rewritings performed are entirely untrusted: generated F⋆ terms are rechecked against the user-supplied specification after the transformation. While this avoids the need for proving that our metaprogrammed call-graph rewriting preserves the semantics of the source program, it however forces reverification of the specialized code. Luckily, the SMT-based proof automation was in most cases sufficient to perform this step automatically when starting from a verified, generic version. Debugging unstable cases was however tedious, as it required printing the generated, specialized program, adapting it to match the F⋆ syntax (the pretty-printer did not exactly generate valid F⋆ code), fixing verification on the specialized program, and finally propagating needed assertions or lemma calls to the generic version to stabilize the proofs.

2.2.2 Platform-Specific Assembly Code with Vale. Project Everest implemented a variety of cryptographic primitives (or the kernels thereof) in verified assembly using Vale (Section 2.1.6). Our goals were both to achieve high performance, and to evaluate and evolve Vale itself. As our techniques, experience, and Vale evolved, we targeted increasingly complex versions of the primitives. We summarize our experience with several representative examples below.

SHA-256. SHA-256 [155] is a ubiquitous cryptographic hash function. Somewhat unusually for a cryptographic primitive, its specification is quite imperative, relying on repeatedly applying a variety of bit-level manipulations of intermediate variables. Nonetheless, optimized implementations, e.g., in OpenSSL, can be quite large, due to careful instruction scheduling, loop unrolling, and other optimizations. Using our original Vale/Dafny implementation, we verified OpenSSL's vanilla assembly implementation on ARM to demonstrate that Vale supports all of the performance tricks that OpenSSL employs, and we verified our own implementation on x86, to illustrate Vale's support for multiple hardware platforms [46]. Our measurements show our ARM implementation matching that of OpenSSL, confirming that we faithfully implemented and verified their optimizations.

We later built an implementation (based on OpenSSL's) for x64 that uses Intel's SHA-acceleration instructions [104]; it too achieves performance parity with OpenSSL [93].

AES. AES [158] is a block cipher used in numerous cryptographic constructions to provide data secrecy and/or integrity. It is used so ubiquitously that Intel added seven dedicated CPU instructions (in the AESNI extension [103]) to accelerate its performance. Specifying the effect of each instruction is complex, since it involves describing large swathes of the AES specification, as well as Intel's XMM extensions for 128-bit registers.

We initially used Vale/Dafny to implement AES-CBC [157] (an encryption mode that provides secrecy but not integrity), following Intel's recommended guidelines for employing the AESNI instructions. This implementation met, and in some cases beat, OpenSSL's AES-CBC comparable implementation, but it is less used in OpenSSL; AES-CBC is not commonly used on the Internet, and when it is, OpenSSL typically uses a version that can process four ciphertexts in parallel, e.g., for multiple TLS connections.

The far more common use of AES is as part of AES-GCM [159], a construction which provides both secrecy and integrity. It is used, e.g., for over 91% of secure web traffic [153]. AES-GCM is quite complex, with the correctness of the GCM portion requiring reasoning about operations over a Galois field. We initially used Vale/F* to implement our own version of AES-GCM [93] in pure assembly, and in a hybrid of C and assembly. At the time, the pure version outperformed the hybrid by 1.5×, and a pure C implementation verified in HACL* by 3,670×. However, it lagged OpenSSL's pure assembly version (written in over 1,100 lines of Perl and C preprocessor scripts) by 6.5×.

We eventually ported OpenSSL's version to Vale/F*. The code is remarkably complex, interleaving instructions for encryption, authentication, and memory prefetching; it processes six 128-bit blocks in parallel to saturate the XMM registers, while also running the encryption 12 blocks ahead of the authentication in an effort to keep the CPU pipeline saturated. Verifying this code exactly as written required significant developer effort. It took non-trivial effort to even understand the existing code; indeed one co-author spent 2 days convinced it was wrong in one corner case, only to discover and prove an invariant covering that case too. Our resulting verified code meets or beats the performance of OpenSSL's implementation.

Curve25519. Curve25519 [28] is an elliptic curve used for key agreement by many modern standards for secure Internet traffic, including SSH, TLS-1.3, WireGuard [76], and Signal.[13] Using Curve25519 requires performing operations over field elements represented using 4–5 64-bit machine words. These field operations are typically bottlenecked by carry-bit propagation, even given Intel's normal support for an add-with-carry instruction, so Intel added support for a second carry chain via the Intel ADX instruction set [117]. In 2017, Oliveira et al. [162] created an implementation using this new support and showed that it resulted in a significant performance boost. Software vendors, including Mozilla and WireGuard, were interested in using the new optimizations, but they became disenchanted when several bugs [77] were discovered through differential testing.

Starting from the implementation by Oliveira et al., we implemented and verified an ADX-based design in Vale, with higher-level operations over those field elements written and verified in HACL* [172]. The implementation was sufficiently modular that the Vale code could be swapped for a verified C-level implementation of the field operations for use on non-ADX CPUs. At the time of publication, both implementations set records [172]. Our pure C implementation beat state-of-the-art unverified C implementations [1], verified C implementation from FiatCrypto [85], and even a popular assembly implementation [63]. Meanwhile, our hybrid, ADX-based implementation beat all known verified or unverified implementations, including the version by Oliveira et al.

Automatic Transformations. In our efforts to match the performance of state-of-the-art industrial cryptographic libraries, like OpenSSL, we focused on verifying exactly that code in Vale. This proved effective in achieving good performance, but in some cases, such as AES-GCM, it created an onerous amount of work, since the code was quite complex, without clean abstractions or separations of concern that would have simplified verification. Hence, we designed a framework [48] for automatically transforming a clean version of the assembly code (e.g., where the encryption instructions are grouped together, followed by the authentication instructions) into the "ugly" but performant version. By verifying that the generic transformation steps preserve the original semantics, we could write and verify the clean code (using 3× fewer lines of proof) but transfer the guarantees to the ugly code. To quantify the performance benefits offered by the ugly code, we evaluated both versions on a variety of different CPUs. To our surprise, on some CPUs (primarily from older generations), the clean version ran faster! We then experimented with a genetic algorithm that attempted to design the fastest version of the code for each specific processor generation, using the verified transformers to confirm that the generated algorithm still provably implemented

[13] https://signal.org.

AES-GCM. This process produced code that could beat OpenSSL on each individual processor by up to 27% [48]. This level of fine-grained CPU-specific specialization would be infeasible in an ordinary development world, where the burden of maintaining so many versions would be tremendous, but with verification-backed automated optimization, such an approach seems quite plausible, although we have yet to deploy these automatically optimized implementations.

2.2.3 EverCrypt: A Cryptographic Provider. From a developer's perspective, cryptographic providers like libsodium [2] offer not simply an ad hoc collection of cryptographic primitives, but rather a comprehensive set of cryptographic utilities united under a clean, unified API. Prior to our work on EverCrypt, no verified equivalent existed.

With EverCrypt [172], we aimed to provide a comprehensive suite of verified cryptographic primitives and constructions, along with modern features such as cryptographic agility and automated multiplexing. An API with cryptographic agility makes it easy for developers to swap between two cryptographic algorithms that provide the same cryptographic functionality (e.g., between the Blake2 and SHA-256 hash algorithms). This is crucial to enable applications to quickly switch away from a cryptographic algorithm that is broken via new cryptanalysis techniques or advances in, say, quantum computation. Historically, a lack of cryptographic agility has created dangerously long windows of vulnerability; e.g, after attacks on SHA1 became feasible, it still took the world over 5 years to migrate to more secure alternatives.

Concretely, EverCrypt's agile API for, say, hashing, accepts an algorithm specifier during initialization, but it returns an abstract state object to the caller. All other API calls operate on that state and take identical arguments, regardless of algorithm choice. Hence, a one-line change to the initialization call enables a new algorithm choice. In practice, we also found that an agile API facilitated cleaner verification. The API's formal guarantees necessarily omit any algorithm-specific details, which means that verified consumers of those APIs have fewer irrelevant facts polluting their proof context.

EverCrypt also performs verified multiplexing, meaning that it automatically probes the CPU's capabilities at library initialization-time, and from then on, selects the fastest algorithm supported on that platform. For example, if the platform includes Intel's AESNI instructions, EverCrypt selects an optimized Vale assembly implementation of AES-GCM; otherwise it falls back to a generic C implementation from HACL⋆. Because both implementations provably implement identical functionality, these selections can occur automatically, without input from the developer. Doing this crucially relied on all our proofs being in the same framework, so that both ValeCrypt and HACL⋆ implementations of the same algorithm offered identical logical specifications, and also on our support for verified interoperability between C and assembly (Section 2.1.6) so that EverCrypt's Low⋆ code could call into ValeCrypt's assembly routines.

As with our other cryptographic efforts, we relied heavily on metaprogramming to author and verify our code once, and then to produce multiple instances of it. For instance, outer parts of our Curve25519 algorithm were generically proven against any implementation of the core field primitives; then, specialized instances were generated "for free" for both the ASM (ADX) and C versions, and those were eventually packaged underneath an abstract layer of multiplexing in EverCrypt. The same goes for other algorithms such as SHA-256.

In the process of creating EverCrypt, we also developed approximately six new optimized assembly implementations in Vale, and 11 new C implementations in HACL⋆, many of which are described above. At the time,[14] many met or beat the performance of state-of-the art implementations, whether verified or unverified.

[14]We believe our implementations remain close to state-of-the-art performance, though we do not continuously benchmark their performance.

We also used EverCrypt in several applications, to illustrate both the usability of the APIs and that EverCrypt's performance gains accrue to the applications too. For example, we developed a highly optimized Merkle tree supporting amortized $O(1)$ insertions. We verified it against EverCrypt's API and also proved cryptographic security, in the sense that finding a collision in the Merkle tree could be provably translated into finding a collision in the underlying hash function. We also found that our Merkle tree was 2.8× faster than Bitcoin's implementation at the time.

EverCrypt also underpins our work on the TLS-1.3 and QUIC record layers (Section 2.4).

2.2.4 Observations. Looking back, our drive for strong performance pushed our tools to evolve to the point where they could handle the complexities found in optimized code produced by practicing cryptographic engineers. We also focused on breadth, with the combination of HACL⋆, ValeCrypt, and EverCrypt providing a broad suite of cryptographic algorithms that can be used in a variety of settings. Indeed, today, a new application that needs a cryptographic library could easily choose to use our verified code with bindings from a variety of languages.

Matching the performance of existing unverified code forced us to grapple with complicated hardware instructions, since an implementation using these instructions can be 8–10× faster than a vanilla C implementation, and 3–4× faster than hand-written vanilla assembly [46]. Using C-level intrinsics can perform better than vanilla C [168], but it requires trusting the compiler, typically with maximally aggressive optimizations enabled.

Verifying so much industrial code made us appreciate the remarkable ingenuity of engineers implementing low-level, optimized cryptographic code. This reinforces our view that we should design verification tools that support such developers as they exercise their ingenuity, rather than try to automatically create our own optimizations for them. We are also amazed that experts can design and implement such intricate implementations that nearly always work correctly, even without the benefit of mechanized assistance. We hope that the verification tools we and others develop will simplify their jobs and perhaps even support further feats of ingenuity by providing a verification "safety net."

2.3 Parsers and Serializers

Protocol standards prescribe specific message wire formats to enable interoperability. Serializing and parsing structured messages to and from a wire format is an essential part of any protocol implementation, and their correctness is crucial to an end-to-end theorem. For security protocols, it is also important that the wire formats are *non-malleable*, i.e., any modification to a wire-formatted message should be detected by the parser, also known as the "unique representation" property.

In prior work on miTLS in F7, a considerable part of the proof effort was spent on manually proving the correctness of ad hoc parsers and serializers. For Project Everest, we decided to solve this problem systematically by developing EverParse, a verified parser and serializer library for non-malleable wire formats, especially those used in TLS. Going beyond miTLS, whose parsers and serializers were programmed in purely functional F#, EverParse aimed for high-performance C code with zero copies.

EverParse was designed as a library of higher-order combinators, as described in a 2019 paper [176]. Combinator parsing has its roots in functional programming [116], providing a higher-order, compositional way to structure parsers. EverParse generalized the combinator parsing approach to produce verified code in low-level languages, by *layering* combinators. Our approach distinguished *specification* combinators, pure functions that *define* the data format specification, and on which proofs of properties such as non-malleability are conducted; and *implementation* combinators which follow the structure of the specification combinators while refining them to efficient, low-level code.

Given a library of specification and implementation combinators, one could build a provably correct low-level parser by composing combinators, with their types ensuring that implementations refine specifications, by construction. This structured approach significantly reduced the cost of doing proofs of specific parsers and serializers, in favor of once-and-for-all generic proofs of the underlying combinators—another instance of our exploitation of the higher-order, generic, dependently typed programming style that F⋆ offers. Additionally, relying on metaprogramming and compile-time specialization, all the higher-order combinators are partially evaluated into low-level code extractable to efficient C code by F⋆ and KaRaMeL.

QD: A Frontend to Consume Notation from RFCs. The wire formats of messages in the RFCs for TLS, QUIC, and others are described in a semi-formal notation for tag-length-value encoded structures. To directly consume this notation, we developed a frontend to EverParse (playfully named QuackyDucky or QD by Nadim Kobeissi who implemented it) to translate the semi-formal notation used in the TLS RFC (as well as some others) into EverParse combinators.

Using QD, we automatically proved the non-malleability of the wire formats for TLS-1.2 and TLS-1.3 and generated validators and parsers for them. EverParse combinators were designed to be efficient zero-copy parsers. That is, one could read elements of a validated message directly out of the input buffer, rather than copying them into a separate data structure. Later, EverParse was used to formalize the wire formats of the QUIC protocol [73], together with manually written parsers and serializers to handle secret-dependent values in a constant-time manner, important for side-channel protection.

EverParse also supports serialization, though it was left to a human to assemble serializer combinators by hand, rather than using QD—this was necessary since we aimed to also support incremental serialization, with full user control over memory management, rather than copying from a high-level message type into a wire-formatted byte buffer. These serializer combinators were used in an interesting variant in DICE⋆ [204], a verified firmware library for IoT devices. To support in-place serialization of variable-length X.509 certificates, an adaptation of the serializer combinators was developed to allow writing data into a buffer from right to left, so that the variable-sized byte length of a variable-length field could be filled in after it was serialized.

ASN.1. Starting in the summer of 2020 and stretching over the course of two remote summer internships at MSR by Haobin Ni, we formalized the ASN.1 data description language. This development was also built on top of the core library of EverParse combinators, with additions to support the various ASN.1 specific features. The work involved distilling the complexity of four ISO standard documents[15] (covering several hundred pages of text) into about 6,700 lines of documented F⋆ code together with the first formal proof that ASN.1's DER produce unambiguous, non-malleable binary formatted messages [156]. We also experimentally validated our formalization by using it to parse nearly 20,000 ASN.1 formatted X.509 certificates and certificate revocation lists found in the wild. However, this development yielded only OCaml code for validators and parsers, rather than low-level C code.

Windows Networking and 3D. EverParse's usefulness was soon recognized to be broader than just for use within verified F⋆ applications. Parsing attacker-controlled inputs is a significant source of security vulnerabilities in all kinds of applications, especially those programmed in memory unsafe languages like C.[16] Recognizing this, working in conjunction with the Windows Networking engineering teams, we designed 3D, a C-like frontend to EverParse that can be used to describe ad hoc, data-dependent binary formats [201]. 3D's "dependent data descriptions" are translated to applications of EverParse combinators yielding verified C code for parsing. Like other parser

[15] https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx.
[16] https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html.

```
typedef struct _TCP_HEADER(UINT32 SegmentLength, mutable PUINT8* data)
{
  PORT            SourcePort;
  PORT            DestinationPort;
  SEQ_NUMBER      SeqNumber;
  ACK_NUMBER      AckNumber;
  UINT16          DataOffset:4
  { //DataOffset is in units of 32 bit words
    sizeof(this) ≤ DataOffset * 4 && //DataOffset points after the static fields
    DataOffset * 4 ≤ SegmentLength //and within the current segment
  };
  UINT16          Reserved:3
  {
    Reserved == 0 //Reserved bytes are unused
  };
  UINT16          NS:1;
  UINT16          CWR:1;
  UINT16          ECE:1;
  UINT16          URG:1;
  UINT16          ACK:1
  {
    AckNumber == 0 ||
    ACK == 1 //AckNumber can only be set if the ACK flag is set
  } ;
  UINT16          PSH:1;
  UINT16          RST:1;
  UINT16          SYN:1;
  UINT16          FIN:1;
  WINDOW          Window;
  CHECK_SUM       CheckSum;
  URGENT_PTR      UrgentPointer
  {
    UrgentPointer == 0 ||
    URG == 1 //UrgentPointer can only be set if the URG flag is set
  };
  //The SYN=1 condition indicates when MAX_SEG_SIZE option can be received
  //This is an array of options consuming the specified number of bytes
  OPTION(SYN==1)  Options[:byte-size (DataOffset * 4) - sizeof(this)];
  UINT8           Data[SegmentLength - (DataOffset * 4)]
  {:act *data = field_ptr; }
} TCP_HEADER;
```

Fig. 7. A top-level specification in 3D of the TCP header format, with dependent fields and constraints. The type OPTION is a user-defined union, whose cases depend on the SYN bit. The last field is associated with an imperative parsing action, which sets the out-parameter data to point to the start of the TCP payload. The 3D tool generates EverParse combinators to parse this format, yielding verified C code.

generators, 3D also supports a notion of "parsing actions," imperative code that can be executed as fragments of the input are parsed. Figure 7 shows a top-level specification in 3D of the TCP header format.

3D also aimed to address the slow verification times of code produced by QD. Whereas QD directly generated applications of EverParse combinators producing a large volume of code to be typechecked by F*, 3D instead uses an interpreter for a deeply embedded language of data formats proven correct once and for all, together with a partial evaluation approach—an instance of compile-time specialization, as described in Section 2.1.7. 3D is used today in many products at Microsoft—we discuss this further in Section 3.

Post-Everest. Work on secure formatting tools continues in the post-Everest ecosystem—we discuss three main strands below.

Comparse. Comparse [214] is another secure parsing framework in F*, with a focus on *specifications* of message formats rather than their efficient implementation. Whereas EverParse comes with a custom domain-specific frontend (QuackyDucky, ASN.1, 3D), Comparse's frontend is F* itself, which allows using the full expressivity of F* to define formats; in most cases, parsers and serializers can be automatically derived using a meta-program. Comparse is parameterized by a `bytes` interface, meaning one can instantiate it with either concrete bytes (to obtain a reference implementation) or DY*'s symbolic bytes (to perform security proofs). This allows obtaining non-confusion proofs for the entirety of the protocol, or finding attacks, such as a recent signature confusion attack in the MLS messaging protocol [213].

PulseParse, CBOR, and CDDL. PulseParse redesigns the implementation-level combinators of EverParse using Pulse, instead of Low*. The use of separation logic in Pulse enabled defining modular combinator-style interfaces for both parsers *and* serialization—in contrast, serialization support in Low*-based EverParse is limited and requires a high-level of manual proof. Additionally, PulseParse provides support for a class of recursive formats, particularly those that can be validated in constant stack space. Using this new foundation, Ramananandro et al. [177] formalize a range of new data formatting standards, including CBOR [47] and a standardized data description language CDDL [42] on top of it—these standards are particularly relevant in security-critical applications for embedded systems such as COSE [189], DPE [207], and others.

Vest. Cai et al. [57] implement a parser-and-serializer generator in verified Rust by compiling a custom data description language (inspired by Nail [18]) to a combinator library in Verus. The Vest DSL is expressive and provides particularly robust support for variant formats (e.g., optional fields or unions that are explicitly tagged, implicitly tagged, or untagged) and dependent formats, both within formats (e.g., for tag-length-value formats) and external to formats (needed to support formats that may depend on a protocol-level state machine). Vest also offers systematic resistance to basic digital side-channel attacks. Thanks in part to an elegant, trait-based treatment of parser and serializer combinators in Rust, Vest verifies parsers and serializers for complex formats, like WebAssembly [105] executables and TLS-1.3 messages, in under a minute.

2.3.1 Observations. Secure parsing and serialization is a surprisingly fruitful area for research in program proof, inasmuch as nearly all applications need to parse and serialize data, and failures in doing so correctly are a major source of security vulnerabilities. Furthermore, one can offer push-button verified code generators for parsing and serialization, allowing users to easily adopt high-assurance code, interfacing with it from verified or unverified contexts.

Other projects, notably DARPA's SafeDocs program,[17] have also highlighted the importance of verified parsing and serialization for application security, particularly for document formats such as PDF, rather than the binary packet and certificate formats that we investigated. The diversity of formats and format languages used in the wild suggests that secure parsing will remain a rich area of research for some time to come. We conjecture that a toolkit for composable format specification, such as EverParse's layered combinators for specifications and implementations, could be a basis of a shared framework in which to build a variety of format-specific frontends and backends.

2.4 Verified Protocols

The miTLS project was a collaboration between Microsoft and INRIA that started around 2010. Its 0.9 version [39] was a verified reference implementation of TLS, supporting TLS-1.0, TLS-1.1, and TLS 1.2, and was implemented in mostly functional F# code with proofs by typing conducted using the F7 refinement type checker [24]. miTLS 0.9 had already been very influential, in serving as a reference implementation of the standard, useful for interoperability and conformance testing. It

[17] https://www.darpa.mil/research/programs/safe-documents.

had also been used to carefully study the protocol, and had led to the discovery of many bugs in other implementations and in the standard itself.

Project Everest aimed to build an efficient, low-level version of miTLS, executable as a C program without a garbage collector, while also supporting TLS-1.3. The goal was to also produce a proof of cryptographic security for the entire protocol, including all its implementation details. We were following a proof methodology developed originally by Fournet et al. [92], though scaled up to support all the details of TLS, and integrated within Low⋆. This methodology enabled game-based cryptographic proofs by typing, relying on type abstraction and modularity to successively rewrite a concrete implementation into a secure-by-construction, ideal functionality. The concrete probabilistic bounds underpinning such game-hopping proofs were out-of-scope of mechanization and were estimated by separate, on-paper formalizations.

TLS Record Layer. Our first contribution was a full formalization and verified implementation of the TLS-1.3 record layer [72] in Low⋆. This work yielded a high-performance implementation of the main streaming, "AEAD" functionality, covering all low-level details including message formatting, and showing that that it provided a cryptographically secure stream. The underlying cryptographic primitives were also verified in Low⋆ and taken from the HACL⋆ project.

QUIC Record Layer. Building on this, we also developed a verified implementation of the QUIC record layer [73] in Low⋆, with a similar end-to-end cryptographic security result, though this time it also included manually written side-channel protections for the parsers and serializers. This record layer was linked with an implementation of the rest of the QUIC protocol developed in Dafny at CMU, though this code was only proven safe, rather than functionally correct or cryptographically secure. The Dafny code was extracted by an experimental C++ backend for Dafny that was developed for this project, relying on reference counting. The resulting system is full-featured and reasonably performant, supporting 2 GB/s of throughput, approximately 79% of the performance of a popular unverified implementation.

TLS Handshake. In parallel with our verified implementations, we also worked on pen-and-paper proofs. For example, we analyzed the downgrade security of the TLS-1.3 handshake protocol [37], resulting in a change in the standard to use specially encoded nonces for downgrade protection. The analysis of the TLS-1.3 key schedule turned out to be extremely difficult. One of the core difficulties in the analysis is *late domain separation*. Concretely, important information such as Diffie-Hellman shares and distinct labels for external PSKs and resumption PSKs are not initially included in the key derivation, but, instead, are included indirectly in the final key derivation when the TLS-1.3 key schedule includes the transcript hash. Thus, *internal* keys are potentially shared between several sessions, which complicates the analysis. Moreover, it requires an *agile* assumption for extracting key values from Diffie-Hellman secrets. See Brzuska [51] and Brzuska et al. [53] for more discussion.

SSP. The complexity of the TLS-1.3 key schedule required us to develop new techniques for modular, code-based cryptographic proofs. We developed *SSP*, a method of writing code-based cryptographic proofs where the code of a game is split into several modules, each of which has its own state [54]. The idea of SSPs, roughly, is to only have two types of game-hops: reduction game-hops and functional equivalence game-hops. Since SSPs structure game code into a call-graph of modules, a reduction to an assumption (which is also specified via a call-graph of modules) consists simply in identifying the sub-graph corresponding to an assumption game within a bigger game, and replacing the "real" assumption game with its indistinguishable "ideal" counterpart [130, 150].

In a significant validation of the SSP technique, we used it to formalize the TLS-1.3 key schedule on paper [53], with the main result proving, in the computational model, that the keys output by the TLS key schedule are secure as soon as *any* of their input key materials are secure. This

was the main result of our protocol formalization efforts, although the pen-and-paper proof was only checked manually. The underlying research became a core component of the PhD theses of Kohbrok [130] and Egger [81].

As it became clear that an analysis of the TLS-1.3 key schedule would be possible in the SSP framework, we began investing in mechanizing the SSP technique, in two concurrent efforts. One effort, in the summer of 2019 by Kohbrok et al. [127], led to the development of an F⋆ library capable of doing SSP of security of relatively simple cryptographic constructions, e.g., Cryptobox [29]. However, a lot more work was needed to scale this development to a usable tool, with Kohbrok et al. noting that with their library "F⋆ struggles when composing packages." With more work, particularly with the use of tactic-based proofs in Meta-F⋆ which had become available by then, one might have been able to scale this further. However, a confluence of factors, including the pandemic which made direct collaboration harder, combined with the F⋆ team's focus shifting in 2020 to the deployment of EverParse in Windows, led to this effort being abandoned.

Around the same time, another effort to mechanize SSP proofs was begun by a group involving several Project Everest members in Europe. This resulted in SSProve [3, 107], a framework for machine-checked SSPs in Rocq, combining ideas from SSPs with our prior work on Dijkstra monads, especially in its generalized form "Dijkstra Monads for All" [147] and its extension to relational logics [148]. A key insight was realizing that the perfect indistinguishability steps in relational program logics would nicely complement the more syntactic package rewriting laws of SSP, and this became a feature of SSProve. The use of Rocq instead of F⋆ reflected the expertise of the SSProve team and also the desire to show the Dijkstra monad approach working in a type theory independently of F⋆. Additionally, since the goal was to prove the security of the *design* of a cryptographic protocol, rather than an implementation like miTLS-fstar, a proof system in Rocq was sufficient. Nevertheless, we had ambitions to port SSProve to F⋆, with the goal of benefiting from better SMT-based proof automation, and the potential to link SSP proofs to verified implementations, though that has not happened yet for a variety of reasons, including shifting priorities across the team as the Project Everest wound down.

Post-Everest. Recently, SSProve was used to prove the security of the TLS-1.3 key schedule and also to connect it to an executable implementation of TLS [41]. Specifically, the work is in the context of Bert13, a fresh implementation in Rust of TLS-1.3. Using Hax, Bert13's code is translated automatically to SSProve. Following our pen-and-paper SSP proof of the key schedule, the authors of Bert13 developed a proof of security of the key schedule of Bert13 in SSProve. Bert13 is also translated using Hax to F⋆ for a proof of runtime safety and parser and serializer correctness, and to ProVerif [43] for authenticity and confidentiality proofs, and post-quantum security. Combining all these results, Bert13, in a sense, finally delivers the end-to-end verified implementation of TLS-1.3 that we had originally aimed for in Project Everest.

Additionally, SSPs have been formalized in EasyCrypt as EasySSP [79], as well as in SSBee [55], the latter being two SSP-specific tools. SSPs have also been used to analyze MLS [52] and secure multi-party computation protocols [56].

Separately, the DY* framework was developed in F⋆ for proofs of security by typing in the symbolic model [32], and used to prove the security of a range of protocols, including Signal [32], ACME [33], and ISO-DH and ISO-KEM [34].

In another line of work, the Owl tool applies information-flow types to cryptographic secrets in order to provide computational security guarantees for protocols designs in an automated fashion [98]. It then compiles the design [192] to a high-performance Rust implementation of the protocol (including parsing and cryptography) that is completely automatically verified for correctness and security preservation.

2.4.1 Observations. Bert13 is a milestone achievement yielding a comprehensive verification result for an executable implementation of TLS-1.3, leveraging a suite of tools, each suited to a specific aspect of the verification. However, a single theorem capturing the security of a protocol as complex as TLS remains elusive. This could be the subject of future work in the field, particularly for researchers drawn to foundational end-to-end guarantees, though one wonders, given the effort it would require, whether such a guarantee would provide any material difference to adopters beyond what Bert13 already offers.

3 Industrial Deployments

From the outset, a key goal of Project Everest was to build verified software and have it deployed in real-world systems. In this we succeeded, though the path to deployment was sometimes circuitous. We start by describing some general factors that contributed to our successes, and then relate some specifics of each of our main deployments, for cryptographic primitives, for parsers, and for protocol code. Much of this code has run in production for several years now, without incident, and as our tools and libraries evolve and improve, more of it continues to be deployed.

3.1 Some Enablers of Success

A key feature of Project Everest was to target the development of security-critical components of larger systems, rather than building an entire verified system. This focus on components was important, perhaps even more than we expected. For instance, rather than a single "drop" of all of Everest's verified software, what we ultimately deployed was more piecemeal, with different commercial customers picking and choosing parts of what we had verified. Despite the pick-and-choose approach, we benefited from tackling a domain where each component had clear, mathematical, functional, and security specifications.

We also targeted standardized systems, since we expected that one implementation of a standard could be swapped with another more easily. This was a good choice; however, standards typically prescribe end-to-end functionality, not necessarily the software APIs involved. Hence, producing code that was also API compatible with a consumer's needs was a challenge.

Furthermore, the software industry broadly recognizes the difficulty of writing correct and secure cryptographic code and (to a lesser extent) parsing code for attacker-controlled inputs. For cryptography, mantras such as "don't roll your own crypto" are well known, and they help encourage the adoption of expert-written code with formal guarantees. Additionally, parsers are tedious to write and there is a long precedent for using parser generators (e.g., yacc or Protobuf), both of which contributed to the positive response to EverParse.

Another crucial choice was to deliver C code (or in some cases assembly) at a level of quality similar to hand-written code, even down to idiomatic formatting and including comments in the generated code. C worked well as a lowest common denominator language, easily integrated with other systems without requiring a significant change in toolchains. Delivering high quality source code allowed consumers without verification experience to study our generated code and to gain confidence in its quality by familiar code-review processes. Further, it reduced concerns around Everest's *bus factor* [65]: even in the absence of the authors, our consumers could maintain our generated code. Providing auto-generated code that is close to handwritten quality has worked particularly well for deployments of EverParse. On the other hand, in some other cases, directly verifying source code written in a mainstream language would have been an easier sell. Indeed, a new effort to port Microsoft's SymCrypt library to Rust and verify the source code via

translation to Aeneas is currently underway, motivated in part by the desire to verify source code directly.[18]

From the beginning of Project Everest, we knew that achieving performance that matched or exceeded the unverified code that we aimed to replace would be critical for deployment. Many of our tool and design choices reflected this emphasis on performance, and it paid off when interacting with potential consumers. Without such strong performance, many conversations would simply have been non-starters.

Finally, many of the integration success stories came via close collaborations, and sometimes via in-person interactions. For instance, the HACS series of cryptographic workshops,[19] as well as various on-site internships, notably at Mozilla, helped to build relationships and to initiate and support deployments. At Microsoft, close collaborations with engineering teams, followed by their advocacy for verified code, have helped significantly. We are most grateful to all our partners in these efforts.

3.2 Cryptographic Primitives

Our first major deployment success was for cryptographic primitives from HACL⋆ at Mozilla. Following considerable engineering investment, HACL⋆ eventually reached a point where clients could "mix and match," and integrate exactly the algorithms they were interested in, by copying a handful of files into their source tree, as opposed to taking a dependency on an entire, external library. This proved to be a major boon for HACL⋆ adoption, as most other libraries require a "wholesale" approach, e.g., building, linking and distributing the entirety of OpenSSL. Mozilla spearheaded the initial integration effort, and now around 17 cryptographic algorithms in NSS[20] comes from HACL⋆ as portable C code, including some SIMD variants. The Vale assembly implementation of Curve25519 is also included in NSS. More recently, post-quantum cryptography from libcrux, verified in F⋆ using the Hax toolchain, was also included in NSS [40].

Jason Donenfeld integrated EverCrypt's Curve25519 implementation into his work on WireGuard in the Linux kernel. Eventually, Python, following a CVE,[21] adopted a large fraction of HACL⋆, starting with hash algorithms, and continuing with all variants of HMAC.[22] Other adopters include the Tezos blockchain[23] and the ElectionGuard[24] voting framework.

While KaRaMeL was always designed with the goal of producing readable and maintainable C code, engaging with consumers forced us to ensure that we really met our goal. For instance, KaRaMeL's pretty printer had to be revised to add additional parentheses for arithmetic expressions, since minimally parenthesized expressions are hard to read and even raise compiler or linter warnings. Additionally, KaRaMeL supported a small DSL to control recombining multiple F⋆ files into a single C translation unit, tweaking visibility of definitions in the process, and eliminating unreachable definitions.

Unsurprisingly, engaging with potential consumers of our code required polishing our code, revising our APIs, or implementing features that we had not initially considered a priority. Working in a proof-oriented language, we could add such features while ensuring that all the guarantees of

[18] https://www.microsoft.com/en-us/research/blog/rewriting-symcrypt-in-rust-to-modernize-microsofts-cryptographic-library/.
[19] https://www.hacs-workshop.org/.
[20] https://firefox-source-docs.mozilla.org/security/nss/index.html.
[21] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-37454.
[22] https://jonathan.protzenko.fr/2025/04/18/python.html.
[23] https://cryspen.com/post/introducing-hacl-packages/.
[24] https://github.com/Election-Tech-Initiative/electionguard-cpp/issues/22.

the code were preserved. It was also important to be able to make such changes quickly, to maintain the engagement of our consumers and to fit their deadlines.

For instance, at some point, HACL⋆ featured four different conventions for error codes, as a result of different developers choosing different conventions for the code they were developing. We eventually managed to settle on a uniform convention with an enum that indicates different error codes, rather than distinguished return values (e.g., `0x0`, `0x01`, `0xffff`).

We had implemented multiplexing APIs in EverCrypt to take care of selecting the best implementation of a given algorithm given the target CPU's capabilities—this was both interesting from a research perspective, and also a requirement for using EverCrypt as a full-fledged cryptographic provider for a protocol implementation like miTLS. However, outside consumers have yet to request this multiplexing layer, since most already have their own version of (unverified) CPU auto-detection. Furthermore, consumers' CPU detection was more comprehensive and precise than what we had, and took into account many more platforms than what we had modeled.

On the other hand, our agile APIs, which offer a unified API for a variety of implementations, without attempting to pick the best one automatically, were more attractive. For instance, landing the Keccak family of algorithms in Python required a new agile, but non-multiplexing API that exposed all six algorithms in this family under one API. We also had to extend our API to support the Blake2 hash function: Blake2 features a low-level API that allows building a tree hash by tweaking many internal parameters of the hash function. We had to extend and (in this case) revise our implementation of this low-level API, because exposing these low-level tweaks required handling many subtle issues.

Our libraries were also initially designed to not fully model memory allocation failures. For deployment in Python, we had to revise our code and proofs to model `malloc` as potentially failing, and to handle and propagate such errors gracefully.

As a general principle, our APIs are defensive, moving static preconditions to runtime checks, to protect against errors in unverified C clients. Tool support to automatically and efficiently wrap APIs with defensive variants, as in secure compilation or gradual verification, could be useful.

3.3 Parser Generators

In 2018, Barry Bond, an engineer on the Project Everest team with long experience in systems development at Microsoft, observed that verified parsers produced by EverParse could help secure attack surfaces in the Windows kernel and elsewhere. In 2019, working closely with MORSE,[25] a cross-cutting offense and defense security team at Microsoft, we followed a data-driven approach using past vulnerabilities to identify a few critical attack surfaces in Microsoft's OS and cloud infrastructure that would benefit from the use of formally verified parsers.

An initial area of focus was Microsoft Hyper-V's Network Virtualization stack, and working closely with the engineering teams, we designed a specification language called 3D for data-dependent, variable length formats, for use by C programmers (Section 2.3). This process of co-development of verification tooling with an engineering team was very productive and led to the creation of a solution that both solved an immediate need, and was general enough to be used to address a class of issues (parsing attacker-controlled inputs) broadly felt in many systems. Not only did the co-development lead to several iterations of the language design, but the specific formal guarantees provided by the toolchain were also refined. Notably, it was important to ensure *double-fetch freedom*, guaranteeing that the C parsers do not read any portions of an attacker-mutable input buffer more than once. Enhancing our proofs to provide this property was a requirement for deployment, and doing so quickly was important to meet product deadlines.

[25] https://www.microsoft.com/en-us/security/blog/author/microsoft-offensive-research-security-engineering-team.

Using a specification language backed by a verified parser generator provided several benefits. Notably, the 3D specification of a format serves as a definitive source of truth, and is easier to understand than imperative C code with subtle corner cases of undefined behavior. It was also seen as a productivity boost: a few lines of refined, data type specification yielded many more lines of tricky, yet provably correct, C code. Further, future issues could be addressed systematically by editing the specification, or, if needed, by revising the code generator. However, successful deployment required surmounting three hurdles: specification discovery, performance, and maintainability.

Specification Discovery and *Testing*. Unlike for some of our other efforts which targeted standards-based software, 3D is used in settings where there is little specification aside from the behavior of an existing software component. Replacing such a component, even if only replacing its parsers, requires careful code reading and testing to discover its specification, followed by specifying its input format in 3D—a process we refer to as *specification archaeology* [114]. Doing so required treading a delicate balance between specifying the desired behavior versus maintaining existing behavior to avoid introducing interoperability regressions. Of course, since the generated code from 3D is proven safe and correct, one can be sure that one does not inadvertently match any insecure behaviors of the existing parser. For Windows Network Virtualization, we specified the formats of more than a hundred different kinds of messages spanning four different protocols, with extensive differential testing to ensure that no unintended breakages were introduced.

Performance. The product teams required that our verified parsers impose no more than an end-to-end overhead of 2% in throughput. This we were able to meet, and even exceed. In some cases, our generated code is faster than previous unverified code, despite containing more checks, usually because our code is designed to be zero-copy, whereas prior code would incur copies in some places, since this was easier for a human to write. As mentioned in Section 3.1, not compromising on performance while delivering security and correctness guarantees is a key enabler of success.

Maintainability. Providing high-quality C code was an important factor for maintainability, since it meant that other engineers trying to understand the use of 3D could use familiar C code as a point of comparison. C code also integrated well with existing profiling and debugging infrastructure, and it provided the option for emergency hot-patching, should the need arise. The generated code had to conform with the coding guidelines of the Windows OS, and for this, KaRaMeL provided specific formatting features. In addition, we relied on clang-format, a source-code formatting tool to enforce various syntactic conventions (indentation, brace positions). For Windows Network Virtualization, we generated around 30,000 lines of verified C code from about 5,000 lines of heavily commented 3D specification. The code has run in production for 5 years now, without incident, although we have revised the specification and regenerated the code as the system has evolved. On occasions when the specification has been revised or refactored for clarity, we have proven using F* that the change introduces no semantic difference. The generated C code has, to date, never been patched directly.

Instead of simply deploying verified parsers for a given format, we actually deployed our entire verified parser generator toolchain in the Windows build environment. Using a build tool extension feature, today EverParse, F*, KaRaMeL, and Z3, are all available for use within the build of every Windows developer. In various OS components, a 3D specification is checked into the source tree, along with the generated C files. Any change to the 3D specification forces the 3D code generator to be run again, regenerating the C files.

Starting with our initial deployments for network virtualization as described by Swamy et al. [201] and in an MSR blog post [178], 3D's functionality and usage has grown to other products, ranging from parsing file formats, to validating pointer-rich data structures. 3D has also been augmented with a symbolic test case generator, 3DTestGen, encoding 3D specifications to SMT and calling Z3 to generate test cases, thus allowing components specified in 3D to not only benefit

from verified parsing, but also from extended test suites and better fuzzing tools. Further, we have built AI agents to assist in authoring 3D specifications [87], using the tests produced by 3DTestGen as feedback to repeatedly refine a specification.

3.4 Protocols

miTLS-fstar was used in several interoperability workshops at the IETF, while the TLS-1.3 RFC was being drafted. Our implementation was revised several times to be compliant with several iterations of the draft. As the QUIC protocol grew in importance and started to be standardized at the IETF, we also implemented an (unverified) version of QUIC in Low*, and this too was used for interoperability experiments.

Around 2018, Microsoft began the msquic project,[26] a cross-platform implementation of the emerging QUIC standard. Our miTLS implementation, notably the handshake and its support for early-data/0RTT-mode was used at Microsoft as the default implementation of the TLS handshake for msquic. We made frequent drops of C code from our implementation to msquic, enabling its development. A limitation was that miTLS-fstar's handshake relied on an unverified region-based memory allocator, so that all data associated with a given TLS connection could be allocated in that region and then torn down when the connection was closed. This stop-gap implementation allowed us to iterate rapidly, while providing value to the msquic effort as it ramped up.

Eventually, our reference implementation was superseded by Microsoft's Schannel library once it gained support for TLS-1.3. Schannel had been and continues to be the default implementation of TLS in Windows. Nevertheless, our work speaks to the value of producing a full-featured version of the standard while it was being drafted. Indeed, work on miTLS-fstar helped uncover a few subtle bugs in the draft TLS-1.3 standard, and also helped with their fixes. For instance, attacks such as LogJam[27] had already demonstrated empirically that older versions of TLS were susceptible to downgrade attacks. While early drafts of the TLS-1.3 standard already provided stronger protection against such downgrade attacks, they were still susceptible to version downgrade attacks in which an attacker downgrades a connection to TLS 1.2 or lower and then mounts one of the known downgrade attacks. This was mitigated in Draft 11 of TLS-1.3 and members of the Everest project developed a cryptographic security proof in [37] that was incorporated into the Everest development. Other reference implementations and analyses of TLS-1.3, including in the symbolic model, also uncovered and led to fixes in the standard [35, 67, 68].

Our pen-and-paper SSP proof for the fixed key schedule [53] shows that the full derivation history of keys is implicitly authenticated. It is noteworthy that the derivation history does not include the identity of out-of-band (i.e., external) PSK holders. This enables a subtle reflection attack [78], if more than two parties (one client and one server) share the same PSK.

After Project Everest, developments in the protocol verification in Everest's ecosystem have focused on messaging protocols, notably on MLS [21]. For example, Wallez et al. [213, 215] formalize the protocol standard in F*, developed in a parametric style that allows both executing the specification as a reference implementation (using HACL*) that interoperates with other implementations, as well as yielding a security model that is analyzed using DY* for a modular proof of cryptographic security in the symbolic model. This effort led to several revisions in the protocol standard, after the analysis revealed security attacks.

4 Challenges and Reflections

In this section, we describe several challenges that we faced, some that we overcame and others that remain. We also share some thoughts on what we may have done differently.

[26] https://github.com/microsoft/msquic.
[27] https://weakdh.org/.

4.1 Coping with Change

Working with an evolving language was an explicit goal of the project, and it allowed us to develop custom tooling and add useful features. However, keeping all our projects working while the language evolved was also a challenge. For instance, HACL⋆ was pinned to an older version of F⋆ in 2017–2018 because the language was changing too quickly. Eventually, we had to upgrade HACL⋆ to the latest version and then keep it up to date, which took nearly a year. These days HACL⋆ is built regularly with the latest F⋆ development version, though it has been a significant challenge to keep a large verified code base always up to date and working with an evolving compiler and libraries. The underlying SMT solver was also pinned to Z3 version 4.8.5 (released in 2019), and only in 2024 did we start to upgrade to the latest Z3 4.13 release.

Despite the evolving language, we managed to quickly grow a large body of code written using Low⋆ and its libraries. However, this was also a mixed blessing. Although we made significant advances on the underlying proving technology (e.g., moving to an approach based on separation logic), transitioning to these newer libraries and DSLs was difficult, since we already had such a large Low⋆ code asset (see Section 2.1.4). Smoothly migrating between libraries, DSLs, and language versions is difficult (e.g., the migration from Lean 3 to Lean 4 was a monumental multi-year effort, using custom porting tools combined with manual code and proof restoration work[28]) and future efforts are likely to face similar challenges.

4.2 Soundness and Trust

F⋆ evolved in support of new language features (better type inference, metaprogramming, universe polymorphism) and libraries, but it also required fixes to soundness bugs we discovered along the way. In total, we have found 26 soundness bugs in F⋆, including three open issues that are undergoing repair. This appears to be similar to experiences in other proof assistants, where unsoundness issues are found and fixed over time. For instance, a quick search of the bug databases of Dafny, Rocq, Why3, Agda, and Lean shows many soundness bugs found and fixed in each tool over the years. Z3 has identified and fixed soundness bugs as well, although we have yet to find a soundness bug in Z3 triggered by the proof of an F⋆ program (we have however reported other kinds of Z3 bugs). One may wonder if soundness bugs in verifiers undermines the entire verification effort—this is a valid concern. However, most soundness bugs are triggered by corner cases and pathological programs which a typical user rarely writes.

Most soundness bugs in F⋆ have been due to implementation bugs (e.g., due to incorrect or missing checks), but three issues were more conceptual. The first is related to a mixture of functional extensionality and subtyping, which was identified by Aseem Rastogi and fixed in 2018.[29] A similar issue was later found in Liquid Haskell, another system that has a similar mixture of subtyping and extensionality, although the fix employed there is different than the solution F⋆ used [210]. Another conceptual issue was related to the injectivity of type formers, which we realized was unsound from prior work on Agda, which Leonardo de Moura had encountered from his work on Lean.[30] This was fixed in 2020, although the fix had to be revisited several times. A third conceptual issue was observed by Gabriel Ebner and involved an incompatibility between classical axioms and monotonic state, the fix for which involved changing the way we represented monotonic predicates.[31]

[28] https://github.com/leanprover-community/mathport.
[29] https://github.com/FStarLang/FStar/issues/1542.
[30] https://github.com/FStarLang/FStar/issues/349.
[31] https://github.com/FStarLang/FStar/issues/2814.

Eliminating unsoundness bugs is a significant challenge for a complex language and implementation, and it is not clear how one might avoid them entirely. Designing a verifier around a small trusted kernel is a well-established approach, used by many proof assistants including Rocq, Isabelle/HOL, and Lean, which helps reduce bugs, though not eliminate them entirely. F⋆ lacks a kernel checker and also trusts both the encoding of VCs to the SMT solver and the SMT solver itself. One might formally verify the typechecker itself, and indeed, a prior version of F⋆ did use a formally certified core typechecker to eliminate bugs in VC generation [194], though it still trusted the SMT solver. Applying this approach to F⋆ as it is today would be a very significant effort, though such efforts are underway for other proof assistants, notably in the MetaRocq Project [193]. We also have formalized various small fragments of F⋆, and recently also implemented a core typechecker for a subset of F⋆, which has helped provide some assurance, but a formalism that covers all of the features of modern F⋆, including extensionality, subtyping, universes, erasure, and non-termination, does not yet exist; we have recently started to work on this. One could also aim to certify proof objects produced by SMT solvers [19, 45, 83]. In fact, the Fine programming language, a predecessor of F⋆, did support proof reconstruction from Z3 [61], though even at the much smaller scale of proofs that were done using Fine, proof reconstruction was slow. Given improvements in SMT solvers and proof reconstruction techniques, revisiting this approach for F⋆ could be worthwhile.

Beyond our core verification tools, we also rely on the trustworthiness of our code-extraction pipeline (as do other verification tools). For us, this means taking steps to ensure KaRaMeL is trustworthy.

In addition to good engineering practices, such as auto-generated visitors, and small, auditable, composable nanopasses (Section 2.1.3), KaRaMeL employs an internal bidirectional type-checker that regularly re-checks the internal representation for typing errors throughout the compilation process; the generated C code also gets re-checked by the C compiler and its diagnostics mechanisms; ultimately, the KaRaMeL developers can check the impact of their changes by looking at a diff for the resulting C code.

Fundamentally, Low⋆ and KaRaMeL are carefully designed to capture the semantics of a (well-behaved subset of) C, which by construction rules out many sources of undefined behavior. As described in Section 2.1.5, we also had a pen-and-paper formalization of compiler correctness.

In spite of all of the above, we still discovered a total of six bugs that were caught by neither our formalization, the internal KaRaMeL checker, nor the C compiler, and had to be identified by debugging incorrect end-to-end tests. The first one was in an implementation of a masking equality function (designed for side-channel resistance), which was a hand-written part of krmllib, the KaRaMeL "standard library." We rewrote and verified this function in Low⋆, and extracted it, rather than hand-write it. The second one was a rare case where an array allocation is of the form `let x = malloc 1 e` (where the initial value `e` refers to a variable `x` *already* in scope), *and* `e` is such that the allocation needs to be compiled as a C `malloc` *followed* by an assignment. Both were handled properly, but the combination initialized `x` with a reference to itself. The third issue relates to a mismatch between implicit integer promotions in C and homogeneous machine integer operations in Low⋆: for instance, `(255uy + 1uy)/2uy` gives `0uy` in Low⋆ (`uy` denotes unsigned 8-bit integers), but `((uint8_t) 255 + (uint8_t) 1)/(uint8_t) 2` gives 128 in C. An elaborate reconstruction procedure was added in KaRaMeL to align both semantics. A fourth issue had to do with writing a 32-bit integer in memory as little-endian bytes, a primitive that is implemented by hand in the KaRaMeL standard library. We wrote `*((uint32_t) ptr) = x;`, forgetting that failing to guarantee pointer alignment is undefined behavior. This was easily fixed by using `memcpy` instead. A fifth issue was more subtle and had to do with lexical scope (in the source Low⋆) and

C99 block scope (in C). In some cases, the lifetime of stack arrays was shorter in C compared to Low⋆, something which was fixed with a dedicated nanopass.

A final issue is that, despite our extensive study of the C standard, we discovered years later that `NULL + 0` was undefined behavior in C. We devised a patch to our Low⋆ model of null pointers and briefly contemplated revising our proofs, but the sheer amount of code that was impacted deterred us from doing so, and we instead ran the generated C code with sanitizers enabled in our test suite. In a last-minute plot twist, and right before this article was going to press, the C standard committee actually approved a revision that makes `NULL + 0` well-defined,[32] to our immense delight.

4.3 Adapting Proofs and Confronting SMT Instability

From the start, a guiding principle of Project Everest was to leverage SMT solvers to reduce the burden of developing software proofs at scale. As described throughout this article, the Z3 SMT solver was our primary proof automation tool, though we carefully mixed SMT solving with other proof techniques, including proofs by reflection, higher-order unification, and tactic-based metaprogramming. Our experience with SMT solving was, on the whole, positive—it is hard to imagine proofs at the scale of ours being done without heavy automation. That said, we also confronted several challenges, including the opacity of SMT solvers and occasional sensitivity to small changes in VCs. We relate some aspects of our experience here, including both some quantitative and qualitative details.

4.3.1 SMT Solving at Scale: The Main Workhorse of Project Everest Proofs. Our reliance on SMT solvers for proofs enabled us to scale quickly to a large verified code base. As a rough estimate, the Project Everest codebase contains more than 600,000 SMT automated proof obligations grouped into around 80,000 separate queries to Z3 (i.e., each query contains several conjuncts to be proven). Although a very coarse measure with a lot of variance, in projects like HACL⋆ that focused on producing executable C and assembly code, our proof-to-code ratio of 2:1 compares quite favorably with other verification efforts, and speaks to the level of automation we enjoyed using a combination of SMT solving and metaprogramming. Other projects, e.g., EverParse, produce no executable code directly, and instead offer tools to generate code from specifications.

Using SMT solvers to automate proofs also helped with proof maintenance and repair [185], a topic that others have studied in the context of interactive proof assistants. SMT-backed proofs typically cope well with changes to a program and its proof structure. Certainly many semantics-preserving but structure altering changes are handled transparently, e.g., reordering the clauses in a conjunct, strengthening a precondition or weakening a postcondition, switching between specification styles such as using a refinement type instead of a precondition, and so on are handled seamlessly, in the vast majority of cases.

The time taken by individual SMT queries varies widely. On modern hardware (AMD Ryzen 97,950X), for a build using *unsat core replay* (that is, pruning the assumptions to facts recorded in unsat cores reported by Z3 from prior proof runs, described below under "Proof replay"), most queries are fast: the median time is 17 milliseconds, and around 3% of total queries are measured to take 0 milliseconds. Only 0.74% of queries take over a second. However, there is a long tail of slow queries. That 0.74% of queries accounts for 50.51% of the total SMT time, coinciding with the anecdotal experience of several users: most queries are fast, but a few hard proofs end up taking most of the time, and are usually the ones that break most often. We present some graphs below showing the distribution of query times (Figure 8). These graphs indicate that the distribution of query times are heavily skewed by a handful of queries.

[32] https://developers.redhat.com/articles/2024/12/11/making-memcpynull-null-0-well-defined.

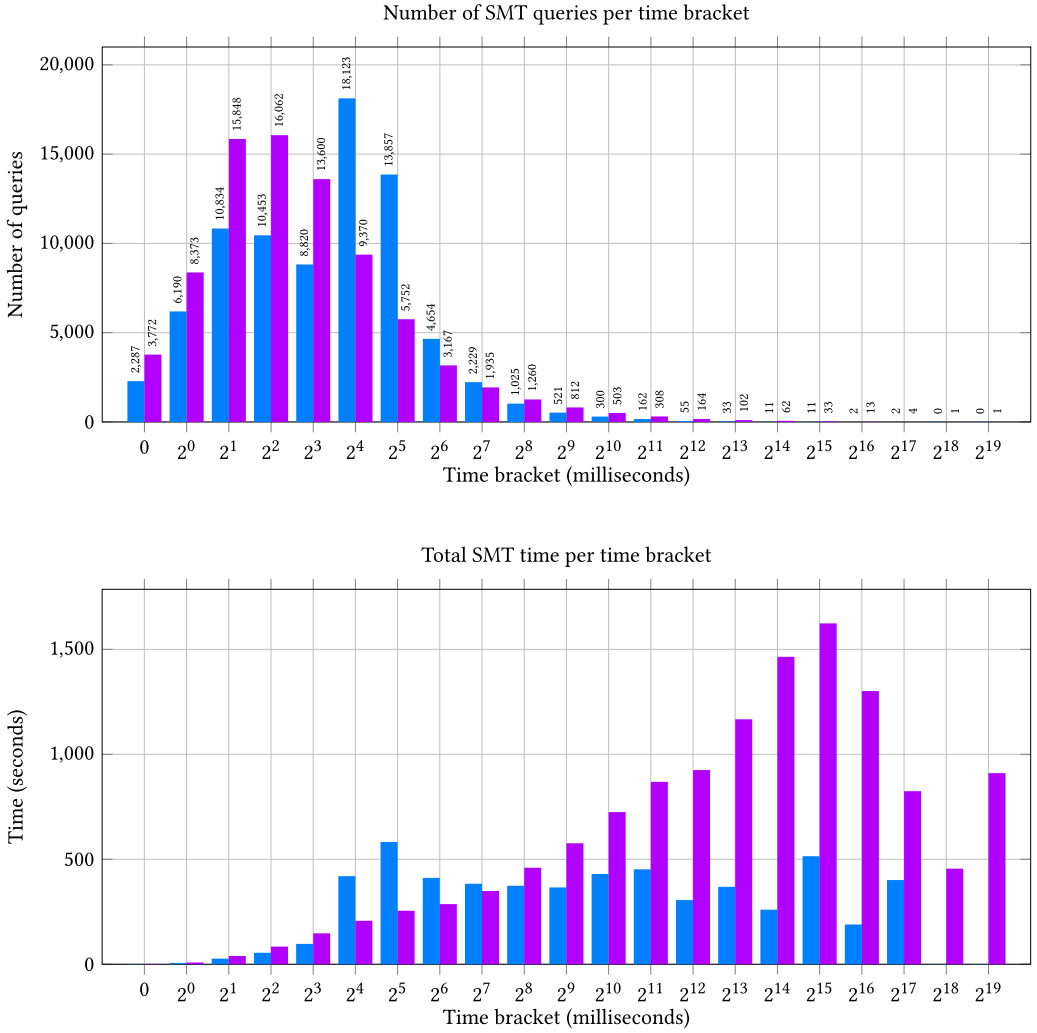

Fig. 8. Some statistics of the SMT queries of Project Everest. A time bracket of 2^i contains queries that take at least 2^i milliseconds and less than 2^{i+1} milliseconds. The first bracket begins at zero milliseconds. The first graph shows the distribution of queries according to their bracket. The second graphs shows the total time taken by each bracket. Blue bars are for a build using unsat core replay, and purple for a build without unsat core replay.

SMT solving accounts for around 27% of the time of a full single-threaded build of Everest, the rest being taken by type-checking, running metaprograms, extraction, and compilation. A parallel build takes around 40 minutes. The stats here are taken from a single-threaded build to reduce noise.

Without unsat core replay, i.e., when checking a proof for the first time, the situation is more extreme. The top 1% of queries accounts for 77% of the total SMT time, compared to 55% for the replay build. The overall proportion of SMT time increases to 48%. The graph makes it clear how the slowest queries dominate even more in this setting. Recording and replaying unsat cores was instrumental in reducing the computational load on users during the development of Project Everest, though they have some drawbacks as described below.

We should note, during the most active development period of the project, common hardware was considerably slower. Also, F* has since gained optimizations (such as context pruning, described below) that have improved SMT performance significantly. Finally, our results are from a recent build using Z3-4.13.3, whereas the project mostly used Z3-4.8.5 until early 2025, though we have not observed a significant difference in performance between the two versions.

4.3.2 Proof Instability. While we found Z3 to be highly effective, it is also not a panacea, and we often also suffered from *proof instability* [109, 227], where tiny semantics-preserving changes (e.g., even simply renaming variables) affect the solver's search heuristics enough to change its result or performance. Similar issues arose when moving from one version of the solver to the next (a major reason why F* was pinned to a 2019 version of Z3 for so long—Section 4.1), or even when running the solver on Windows versus macOS.[33]

Zhou et al. [226, 227, 229] have studied proof instability in depth, including in projects that used F* and Dafny. They report that between 2% and 5% of proofs can be unstable in the various projects, although these results are likely optimistic, since they measure completed projects, rather than the instability that developers encountered and overcame during development. Our experience across a typical build of Project Everest is that the number of unstable proofs is considerably lower, since in a codebase with 600,000 SMT-automated proofs, an instability rate of even 0.001% can result in half a dozen build failures for no apparent reason, which is not our experience. However, this low rate can also be attributed to all the effort expended to make proofs stable, since most proofs in our main branches have survived through tens of thousands of commits. Newer proofs that have yet to be refined do fail more often, a significant problem that can frustrate proof maintenance and evolution efforts. Particularly painful is when a change one developer makes triggers an instability-related failure in another developer's code. Negotiating the technical and social aspects of such failures (e.g., assigning blame for build breakage) can be challenging.

Specifically, an unstable proof manifests as an F* program whose VC was once deemed `UNSAT` (meaning the program is proven) to instead fail with `UNKNOWN`. F*'s encoding to Z3 makes ample use of quantifiers and, as such, the problem of entailment checking is undecidable. As such, on proof failure, Z3 seldom reports `SAT`, and instead reports `UNKNOWN` with a partial model useful for an error report, or can simply time out. Time-outs are particularly frustrating, since in such cases F* cannot report a well-localized error.

To control the time-out behavior, Z3 exposes a logical resource limit, which is machine-independent and deterministic. F* users can set this resource limit on a per-proof basis when invoking Z3. Proofs with a large resource limit allow Z3 to search for a proof for a long time; but, such long-running proof searches are also sensitive to internal search heuristics. As such, we generally preferred proofs with smaller resource limits, aiming to find a balance between predictability and automation, though striking the right balance is not always easy, especially for newcomers. Committing a proof with a large resource limit is usually a bad idea, and it is likely that such a proof will soon break and have to be revisited—a few large resource limit proofs remain in various projects and have become notorious among the maintainers, and failures in these proofs are the usual suspects in a broken build.

We attempted to address proof instability in the following ways.

- —Abstraction: We heavily used F*'s module system, as well as various controlled opacity mechanisms, to hide definitions from the SMT solver.
- —Structured proofs: For families of proofs that follow a similar shape, it is sometimes useful to design combinators or even special syntax to create them. One instance of this was introducing

[33] For a recent issue reporting platform-dependent behaviors, see https://github.com/Z3Prover/z3/issues/7859.

"calc" proofs [141], where an equality (or more generally any transitive relation) is proven by a series of steps. Every step is isolated from the others, which greatly improves SMT performance and helps to make the proof human-readable.

—Proof replay: Once the solver finds a proof, F⋆ provides a feature to record the solver's "unsat core," the subset of the original axioms that allowed the solver to derive `UNSAT`. F⋆ can then use this core to replay a proof on subsequent runs—this helps minimize the sensitivity to search heuristics and also improves verification-replay times. However, recording and replaying unsat cores comes with other challenges, including complexity in the build system, storing additional files with unsat cores in the repository which need to be refreshed periodically, as proofs and tools change. Additionally, for various technical reasons, a small fraction of unsat cores cannot be replayed[34] and proof search has to start from scratch every time. Fully reconstructing the SMT proof in F⋆ would be an interesting but challenging direction to pursue, e.g., following techniques used by tools like Sledgehammer for Isabelle [44].

—Context pruning: F⋆'s default behavior is to encode all pure definitions in the dependency graph to the SMT solver. This can lead to very large SMT files, sometimes with premises numbering in the hundreds of thousands. F⋆ provides a small language of options to selectively remove some premises from the SMT solver's context—expert developers used this feature in places, though this was often a last resort to stabilizing a proof. More recently, with learnings from Verus [226], F⋆ has taken a more systematic approach to context pruning, using a syntactic reachability criterion to slice away a large fraction of the premises from the SMT solver, with significant improvements in proof performance and stability. Context pruning is on by default in F⋆ today, and many projects have found it to both improve proof performance as well as to provide a similar level of proof stability as record-and-replay of unsat cores.

—Selective use of theories: To promote proof stability, some projects disabled the use of the nonlinear arithmetic theory in the SMT solver—this theory is known to sometimes be both inefficient and unstable [89, 109, 139]. The Vale and Zeta [16] projects both adopted this approach. Other projects disabled non-linear arithmetic selectively, e.g., in HACL⋆, disabling non-linear arithmetic was seen as too onerous for newcomers, but expert-authored parts of the library have disabled non-linear arithmetic reasoning in Z3.

—Uncovering instability: We added options to F⋆ to retry proofs multiple times, sometimes with different random seeds, and to report results from these attempts. Proofs that do not consistently succeed encourage the user to focus on spelling the proof out in more detail, until the proof stabilizes.

—Not SMT only: Finally, for proofs that are not efficiently and stably automated by SMT, F⋆ provides other means to do proofs, including normalization and tactics—this places a greater burden programmer, but at least provides a way to get the job done. That said, choosing a proper combination of dependent types, SMT solving, normalization, and tactics, using each where they work best, is a big part of what makes a good verification design.

SMT proof instability was exacerbated by the interaction among many different concerns when doing a complex proof. One anecdote is from HACL⋆, which provided "streaming" APIs to allow clients to feed an arbitrary number of bytes into an algorithm, relying on a library-managed, internal, long-lived piece of state that stores data in a buffer until a full block can be flushed into the underlying block-based algorithm. This API required simultaneously reasoning about modifying heap state, while allocating temporaries on the stack and juggling multiple pieces of data (key state, block algorithm state, user data, output array). While initial experiments were promising, over time, this piece of code turned into a frustrating effort where proofs would regularly break because

[34] https://fstar-lang.org/tutorial/book/under_the_hood/uth_smt.html#hints-that-fail-to-replay.

of proof instability. Ultimately, this piece of code was almost entirely rewritten in a very manual style, performing memory reasoning entirely by explicitly calling lemmas, disabling non-linear arithmetic, pruning the context to remove problematic premises, and adding verbose intermediate assertions in the middle of large stateful functions that required writing and maintaining complex descriptions of intermediary states.

Further research into proof stability and transparency into the SMT solver's proof search is an important area of ongoing [10, 226, 229] and future work.

4.4 Visibility of Proof States

SMT-based proof assistants, including F⋆, Verus, and Dafny, by default, encode the VC for an entire procedure into a single SMT query. If a query fails, a typical mode of interaction is for the user to assert intermediate facts, checking to see what the solver is able to prove, and then focusing their attention on elaborating the parts of a proof that the solver has trouble with. Although the final proof can be terse, sometimes the process of finding that proof can involve many steps of exploration not often preserved in the final artifact. Performance problems with proof search in the solver requires understanding how quantifier instantiation works, and to wield a variety of profiling tools that Z3 provides.

This is a very different mode of interaction than what is offered by a tactic-based interactive prover, where at each step of the proof, the context and goal are explicitly visible to the user. We often had visiting researchers or interns who were used to the proof-state interaction of tactic-based provers for whom the style of asserting intermediate facts was alien, at least at first. Of course, debugging the behavior of ad hoc automation with tactic scripts also has its own set of distinct challenges.

Recent developments in Pulse aim to address the proof-state visibility problem, by employing a symbolic execution with several small SMT queries, rather than a single query for an entire procedure. Symbolic-execution-based verifiers for separation logic are well known (e.g., Smallfoot [25, 26], VeriFast [119], and Viper [82] all use symbolic execution), though Pulse's approach is integrated with F⋆'s dependent type system and is designed for interactive use. That is, Pulse presents the proof state at each step in full detail to the programmer, similar to a style called live verification [101] that others have proposed. Pulse also aims to improve proof stability, by ensuring that the proof of a prefix of a procedure body, once completed, cannot be affected by the suffix. Aeneas [111] has taken another angle on this, by translating Rust programs to pure functions in Lean and then conducting interactive tactic-based semi-automated proofs.

4.5 Requiring Deep and Broad Expertise

Doing protocol proofs at the level of detail of implementations was a significant challenge, for several reasons.

We aimed to provide cryptographic security guarantees about the actual low-level C code, rather than just a model of the protocol—a very ambitious goal. We aimed to do this effectively in a single proof step, mixing cryptographic ideal functionality with the executable code, and relating them by a form of information-flow typing encoded using type abstraction.

Doing such a proof required team members to master the protocol standard; the low-level system design; the Hoare-style program logic and how to write scalable, stable proofs while mastering the various proof styles that F⋆ offers; and finally, one had to also understand the cryptographic proof on paper and its manifestation in the code. No one person in the team was expert in all of this, and the exchange of information and ideas took a lot of effort, teaching, and delegation from one part of the team to another.

We believed then that it was too costly to introduce additional abstraction layers, e.g., to first relate the executable code to a simpler idealized model, and then to carry out a cryptographic proofs on the model. In hindsight, this might have helped in separating concerns and allowed us to make better progress. We did apply this refinement methodology in other parts of the project, though not at the scale of the TLS protocol itself. For example, HACL⋆ proofs involved a low-level functional specification related to the actual code, and then a second high-level specification proven to abstract the low-level one; EverParse adopted a methodology of layered combinators; and some post-Everest work applied a layered approach to protocol security as well [112]. Later work on Owl [98] also explicitly separates cryptographic reasoning from implementation reasoning.

We also took some steps to onboard people who were experts in cryptography, though not in program proof. For instance, HACL⋆ includes a set of libraries built on top of F⋆'s standard library, offering a smaller surface area of functionality for a newcomer to study, and supporting various usage patterns that were idiomatic in HACL⋆, though such patterns were tailored to one specific style of code—both their strength and weakness.

4.6 Communications and Perceptions

Project Everest was a very focused effort, with tightly woven collaborations throughout the core team. The arc of the project and the many threads of work, both in program proof methodologies and their applications to systems and cryptographic applications, had an internal coherence that was apparent to many of the core team members. However, from conversations with others, we are aware that the view from the outside was less clear.

Our main vehicle for communicating results was peer-reviewed, conference publications: we count 27 papers published between 2016 and 2021 as the core research documentation of our work, with many other papers building on this work since then. Forming a complete picture of our work from these papers is challenging. Even the reviewers of this article have remarked "*Given the sheer number of papers published over the years, it is often difficult for an outsider to dive into this ecosystem*" and "*there are so many interesting new methodologies that it can be hard to keep track of them.*" Indeed, one purpose of this article is to provide more of a global view of our work.

Our outreach efforts were also focused primarily on potential industrial adopters of our work, including at Microsoft, Mozilla, and elsewhere. In contrast, the DeepSpec project,[35] an NSF Expedition on program verification at several academic institutions, was much more explicit in its outreach to academics, e.g., running several summer schools. In combination with nearly two decades of Rocq-based curriculum in software foundations,[36] this has made many graduate students familiar with Rocq and interactive proof. In comparison, F⋆'s industrial focus has meant that it remains less widely used by academics and students.

The stability of Rocq as a platform has also made it easier to build and maintain educational resources. As F⋆ now evolves less rapidly than before, we have also started to develop more educational material, including a more comprehensive online textbook[37] together with some efforts at teaching F⋆ in university courses.

Another aspect is perhaps that the academic programming languages community has been focused on themes from which Project Everest explicitly departed. For instance, compiler verification has been a major theme academically (e.g., with CompCert [142] and its offshoots), but we did very little in this direction, focusing instead on source-level verification and expecting that our work would eventually compose with compiler verification technology.

[35] https://deepspec.org/main.
[36] https://softwarefoundations.cis.upenn.edu/.
[37] https://fstar-lang.org/tutorial/.

The academic community has also focused more on foundational results that minimize the TCB, whereas our focus was more pragmatic and focused on automation at scale with transpilers and SMT solvers. While foundational results are possible to achieve for end-to-end results [86], and would eliminate at least some of the bugs we encountered in transpiling Low★ to C (Section 4.2), the proof effort involved, even for protocols significantly simpler than those Project Everest tackled, is much higher. Nevertheless, despite the greater TCB, we aimed to provide the strongest forms of application-specific guarantees, including functional correctness with side-channel resistance for a wide variety of cryptographic algorithms in HACL★; formally verified VC generators for Vale, coupled with a certified taint analysis for side-channel resistance and a verified model of C-assembly interoperation, applied to hand-optimized assembly implementations of cryptographic algorithms in ValeCrypt; proofs of security in the computational model for the TLS and QUIC record layers; SSPs and SSProve for computational proofs of cryptographic protocols; EverParse for proofs of non-malleability for message formats, manually proven constant-time guarantees for parsers and serializers of secret-dependent values in QUIC, and double-fetch freedom for parsers in 3D; foundational models for CSL in Steel; and so on.

Ultimately, Project Everest demonstrated that a focused team of experts could deliver and maintain a large body of production-quality verified code, with a balance between foundational results and practical concerns. Doing so required developing many novel tools and methodologies, and a significant investment in engineering and infrastructure. Coupling this with more of a focus on community building might have helped with greater visibility and adoption, something we hope to do more of in the future as our work evolves and matures.

5 New Directions

Drawing lessons from Everest, and responding to new trends in computing, we describe several ongoing efforts and speculate a little on the path ahead for building and deploying high-assurance systems at scale. We focus primarily on some topics being actively explored by teams involving members of Project Everest.

5.1 High-Assurance Systems Programming in Rust

Rust's emergence as a convincing safer alternative to C and C++ for systems programming is a prominent industrial trend, with many major companies,[38] open source projects,[39] and governments[40] announcing initiatives investing in Rust.

The program-verification research community has responded to this trend with many projects now targeting Rust programs, from a variety of angles. Due to its strong ownership discipline, safe Rust programs can, in many cases, be reasoned about without resorting to an explicit notion of heap and addresses. The design space of such tools is large, offering guarantees ranging from bug-finding by bounded model checking to functional correctness. We cover three main approaches: custom verifiers for Rust; tools that translate Rust into other proof assistants; and, finally, frameworks that emit verified Rust code.

Custom Verifiers for Rust. Tools like Prusti [219], Flux [138], and Verus [136, 137] and several others [74, 97, 209, 232] analyze annotated programs in custom logics designed specifically for Rust. For instance, Verus leverages Rust's ownership discipline through a logical encoding that it sends directly to Z3, aiming to provide an experience similar to Dafny or F★. However, due to Rust's

[38] Amazon [152], Google [99], and Microsoft [151].

[39] Linux [174] and Mozilla [154].

[40] For instance, the Biden administration's National Cyber Security Strategy [217] advocates for the use of Rust and other memory safe programming languages, while also calling out Project Everest's work on verification for securing the software supply chain. DARPA's TRACTOR program [69] also aims to stimulate research into translating C code to Rust.

ownership, its logical encoding does not explicitly encode a mutable heap, resulting in verification times that can be orders of magnitude faster [136]. Verus also supports reasoning about concurrency, though shared memory does add some additional proof obligations. Verus has already enabled the verification of complex systems, ranging from OS microkernels [62], security modules [230], cluster management controllers [196], programming tools such a concurrent memory allocator [136], a library for NUMA-aware concurrent data structure replication [136], a crash-safe storage system [136], and even a library for verified parsers and serializers [57] similar to EverParse. It has also found use in several production-level services at Amazon [220].

Translating Rust to Proof Assistants. The Hax tool [38] targets the verification of sequential Rust programs (decorated with specifications and proof hints) into a variety of proving backends, including F* and Rocq [106]. The F* backend targets a pure fragment of F*, with SMT-backed proofs, whereas the Rocq backend targets an imperative DSL called SSProve [107] embedded in Rocq, with interactive proofs for cryptographic protocols in the computational model. There is also a backend to Proverif [43], for security proofs of cryptographic protocols in the symbolic model. Hax has also been used to verify a range of security protocols and cryptographic code, including post-quantum cryptographic algorithms now deployed by Mozilla and OpenSSH, and a symbolic proof of security for TLS-1.3 via translation to Proverif [38]. As mentioned previously (Section 2.4), in a substantial end-to-end proof [41], Hax has been used to prove the correctness and security of Bert13, a Rust implementation of TLS-1.3, including automated proofs of runtime safety and parsing correctness via translation to F*; a proof of computational security for the key schedule via translation to SSProve (a transcription of the pen-and-paper SSP from Project Everest [53]); authenticity, confidentiality and post-quantum security via translation to Proverif; and using verified cryptography from HACL*—achieving, in a sense, the protocol level implementation proof that was left unfinished in miTLS-fstar.

Like Hax, Aeneas [111] also translates sequential Rust code into other proof assistants. However, it targets the verification of *unannotated* Rust code. It leverages Rust's ownership types through a functional, executable encoding of sequential Rust programs that can be sent to a variety of proof assistants, notably Lean, but with support also for F*, Rocq and HOL4. This allows a proof engineer to specify and reason about the behavior of the translation output, enabling a clear division of labor between software engineer and proof engineer. Aeneas has invested heavily in the Lean theorem prover, allowing the proof engineer to benefit from an interactive mode first and foremost, then defer to automated tactics and proof-search procedures (`Aesop`, `Lean-Auto`, `Duper`) as a controlled way to automate proofs. In an ongoing effort, Microsoft's cryptographic library, SymCrypt, is being rewritten in Rust, with Aeneas used to verify functional correctness.

Extracting Rust from Proof Assistants. Continuing the design spirit of languages like Low*, one can also develop verified code in languages that model a subset of Rust, and then extract Rust code after verification. Notably, KaRaMeL has evolved to include a Rust backend [94], which reconstructs structured, safe Rust code out of KaRaMeL's C-like input language. Low*, Steel, and Pulse can now be compiled to Rust, though one must program in a source discipline that mimics Rust's borrow checker to ensure that the generated Rust code compiles. For instance, if the programmer made use of aliasing permitted in, say, Pulse (e.g., doubly linked lists, expressible safely in Pulse but not in Rust), the resulting program could fail to borrow-check in Rust. Using this capability, EverCBOR [177] develops verified parsing and serialization tools for a range of format standards in Pulse, and it extracts the code to both C and Rust, supporting consumers from both ecosystems. Additionally, using Rust support in KaRaMeL, HACL* can now be extracted to Rust, enabling consumers to adopt Rust code directly, instead of C code with Rust wrappers. One might expect safe Rust's additional runtime checks on array accesses to be costly; however, our preliminary investigation indicates that this overhead is not substantial.

5.2 Tools for Cryptographic Applications

Formal analysis techniques for cryptographic applications remains an important area of research: we briefly mention a few themes.

Side Channels. The emergence of speculation-based micro-architectural side channels [128, 146] has led to a flurry of activity to find and defend against such attacks [113, 161, 211], including in cryptographic settings [131]. Side channels in cryptographic implementations continue to be an important problem, independently of speculation [31].

SSPs. As discussed in Section 2.4, tools such as SSProve and SSBee implement the SSPs formalism. However, one important limit of both SSProve and SSBee are *statistical* indistinguishability arguments, which they cannot prove, but rather need to use as an assumption. Those, in turn, can be precisely argued about in EasyCrypt [23]. A promising future direction is to export assumptions from SSProve and SSBee to EasyCrypt, so that they can be verified in EasyCrypt. Another useful combination might be to use SSBee for fast prototyping of invariants for use in EasyCrypt; invariant search in EasyCrypt can be time consuming, since trying each candidate invariant may require substantial re-writing of the proof attempt. A third approach, pioneered in EasySSP [79] is to directly perform SSP analyses in EasyCrypt. State-separation is an instance of using information-flow control to structure cryptographic proofs, although not the only one. For example, Owl [98] exploits an information-flow type system to track dependencies, or lack thereof, among the components of a protocol as a basis for modular proofs.

Post-Quantum Cryptography. As the industry transitions to the use of post-quantum cryptographic primitives, there is an opportunity to base the next generation of cryptographic implementations on formally verified implementations. Many organizations have seized this opportunity, working closely with formal verification experts, including from Cryspen [125], a cryptographic verification startup built around high-assurance cryptographic software inspired by HACL* and related technologies.

5.3 AI-Assisted Specification and Proof

Another prominent trend of the past few years has been the emergence of generative AI, including in combination with proof assistants. While AI-assisted formalized mathematics has received considerable attention, notably in Lean [181, 216, 222], with prominent mathematicians advocating for AI copilots to assist experts in building large proofs,[41] here we focus specifically on AI as it pertains to program proof.

Especially as the use of AI programming assistants sees widespread use alongside concerns regarding the trustworthiness of AI-generated code [165], there is growing interest in having AI tools target proof-oriented programming languages. This promising direction, both to ensure that AI-generated code is correct, as well as to reduce the human cost and expertise to produce code with proofs, has spurred a growing sub-field studying the use of AI models for specification and proof engineering [60, 90, 123, 133, 145, 166, 175, 188, 195, 205, 221, 222].

Although there have been few controlled studies of users of proof assistants (though a few such studies are beginning to emerge [186]), anecdotally, we remark that AI assistants help in lowering the barrier to entry for proof assistants. For instance, a general purpose chatbot can already produce common textbook algorithms and proofs in many proof languages, allowing newcomers to explore how familiar idioms are expressed and proven in languages that may otherwise be unfamiliar. Code authoring assistants, such as GitHub CoPilot, are trained on code from many proof-oriented languages and some of the present authors (experts in their tools) use these tools as part of their

[41] https://www.scientificamerican.com/article/ai-will-become-mathematicians-co-pilot/.

daily workflow for code completions and to automate simple refactorings of code backed by proofs in F* and other languages.

Software Proofs. Geared for software proofs, the Project Everest codebase is a valuable artifact for training and evaluating the progress of AI tools in proof engineering. Indeed, Chakraborty et al. [59] assemble a dataset of 940,000 lines of code and proof in F*, consisting largely of Project Everest code, with various useful metadata (e.g., proof dependencies), proof-checking, and benchmarking tools. They also prompt **large language models (LLMs)** and fine-tune various smaller models to synthesize F* code and proofs from formal specifications. Chakraborty et al. categorize their results according to precision of the specification, ranging from proofs of lemmas (where any type-correct proof will do), to dependently typed definitions (whose types may capture functional correctness, but could in some cases be less precise), to simply typed definitions (whose specifications are relatively weak). Using their best model, Chakraborty et al. report successfully synthesizing proofs for around 35% of lemmas, 41% of dependently typed definitions, and 61% of simply typed goals.

Intent Formalization and Validation. AI-assistance for authoring specifications is also an important area of work. For instance, researchers have begun exploring how LLMs can be used to synthesize specifications from natural language [84] and, more generally, use AI to help users formalize computational intent [132]. Ensuring that AI formalizations are accurate reflections of a user's intent is an important concern, with researchers proposing various techniques to judge the quality and consistency of the proposed formalization. For instance, the Clover by Sun et al. [195] relies on AI to paraphrase the formalized intent back to natural language as an added sanity check, and report promising empirical results, a paradigm of AI-based validation referred to as using an "LLM as a judge."

Symbolic Tools to Validate AI-Generated Specifications. Aside from AI-based validation, we believe that coupling specification synthesis tools with symbolic checkers and correct-by-construction code generation is a powerful way to leverage the ability of AI models to summarize natural language and to translate between representations in seemingly creative ways, while also minimizing trust in AI code generators. For instance, 3DGen is a suite of AI agents developed by Fakhoury et al. [87] to translate format descriptions in RFC documents to formal specifications in 3D, an input language for EverParse. Using feedback from symbolic tools (such as type checkers and test case generators), these agents can automatically revise their output until they arrive at a 3D specification that matches a user's classification of a large number of symbolically generated test cases. From there, EverParse's verified code generator takes over and can produce efficient and safe C code, provably implementing the synthesized specification.

In summary, the confluence of AI with proof assistants is an exciting and very active new area of research. The difficulty of program proof has for long been a limiting factor in its adoption and broadening access to program proof is perhaps the most pressing problem the field faces. Generative AI offers a new attack on this problem, which when accompanied by steady improvements in the underlying proof tools, could significantly reduce the expertise needed to develop proofs of software. Initial steps in this direction are encouraging, though much remains to be done before this vision is truly realized.

6 Conclusions

Project Everest brought together a large and diverse team of researchers to work intensely together for half a decade on a challenging and practical problem: deploying provably correct and secure implementations of communication software crucial to the Internet's infrastructure. It was, we believe, unique for its focus on co-developing a proof-oriented programming language with a suite of applications. With a strong focus on performant, low-level code in C and assembly, we

prioritized producing components for deployment in the existing software ecosystem, and in this, we succeeded in large measure.

The arc of the project evolved over its five-year course, growing in scope in some regards to prioritize proofs of components that we did not originally target. Conversely, there were parts of our initially proposed plan that we did not complete, reflecting the realities of the shifting priorities and focus of a large team spanning industry and academia, and of the shifting trends of the computing industry itself. Many offshoots of Project Everest thrive today, some building directly on the tools and software we produced, others learning from our mistakes and exploring new approaches to program proofs at scale.

In summary, we believe ambitious, collaborative efforts such as ours challenge the community to both advance the state of the art and impact the current practice of secure software development, though it requires a team motivated to have industrial impact with their research, backed by the organizational support to take multi-year, risky bets. We are grateful to all our institutions and funding agencies for supporting long-range foundational research, and encourage others to take on similar challenges.

References

[1] curve25519-donna. 2008. Implementations of a Fast Elliptic-Curve Diffie-Hellman Primitive. Retrieved from https://github.com/agl/curve25519-donna

[2] The Sodium Crypto. 2013. Library (Libsodium). Retrieved from https://github.com/jedisct1/libsodium

[3] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. 2021. SSProve: A foundational framework for modular cryptographic proofs in Coq. In *Proceedings of the 34th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 608–622.

[4] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New results on instruction cache attacks. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*.

[5] Danel Ahman, Cédric Fournet, Cătălin Hriţcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2017. Recalling a witness: Foundations and applications of monotonic state. *Proceedings of the ACM on Programming Languages* 2, POPL (December 2017), 1–30.

[6] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra monads for free. *ACM SIGPLAN Notices* 52, 1 (January 2017), 515–529.

[7] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John Mitchell, and Dawn Song. 2010. Towards a formal foundation of web security. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, 290–304.

[8] 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)*.

[9] Thorsten Altenkirch and Conor McBride. 2003. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*.

[10] Daneshvar Amrollahi, Mathias Preiner, Aina Niemetz, Andrew Reynolds, Moses Charikar, Cesare Tinelli, and Clark Barrett. 2024. Using normalization to improve SMT solver stability. arXiv:2410.22419. Retrieved from https://arxiv.org/abs/2410.22419

[11] Cezar-Constantin Andrici, Danel Ahman, Cătălin Hriţcu, Ruxandra Icleanu, Guido Martínez, Exequiel Rivas, and Théo Winterhalter. 2025. SecRef*: Securely sharing mutable references between verified and unverified code in F*. arXiv:2503.00404. Retrieved from https://arxiv.org/abs/2503.00404

[12] Cezar-Constantin Andrici, Ştefan Ciobâcă, Catalin Hritcu, Guido Martínez, Exequiel Rivas, Éric Tanter, and Théo Winterhalter. 2024. Securing verified IO programs against unverified code in F★. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2226–2259.

[13] Cezar-Constantin Andrici, Théo Winterhalter, Cătălin Hriţcu, and Exequiel Rivas. 2022. Verifying non-terminating programs with IO in F*. *Presentation at the 10th ACM SIGPLAN Workshop on Higher-Order Programming with Effects (HOPE)*.

[14] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On subnormal floating point and abnormal timing. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[15] Andrew W. Appel. 2015. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems* 37, 2 (2015), 1–31.

[16] Arvind Arasu, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Aymeric Fromherz, Kesha Hietala, Bryan Parno, and Ravi Ramamurthy. 2023. FastVer2: A provably correct monitor for concurrent, key-value stores. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23)*. ACM, New York, NY, 30–46.

[17] Arm. 2025. NEON Instructions. Retrieved March 2025 from https://developer.arm.com/documentation/dui0473/m/neon-instructions

[18] Julian Bangert and Nickolai Zeldovich. 2014. Nail: A practical tool for parsing and generating data formats. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 615–628.

[19] Haniel Barbosa, Clark Barrett, Byron Cook, Bruno Dutertre, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, et al. 2023. Generating and exploiting automated reasoning proof certificates. *Communications of the ACM* 66, 10 (September 2023), 86–95.

[20] 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*.

[21] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. 2023. The Messaging Layer Security (MLS) Protocol. RFC Editor. Retrieved from https://www.rfc-editor.org/info/rfc9420

[22] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. 2022. Hybrid Public Key Encryption. RFC Editor. Retrieved from https://www.rfc-editor.org/info/rfc9180

[23] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. 2011. Computer-aided security proofs for the working cryptographer. In *Proceedings of the International Cryptology Conference (CRYPTO), Organized by the International Association for Cryptologic Research (IACR)*.

[24] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems* 33, 2, Article 8 (2011), 1–45.

[25] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO '05), Revised Lectures*. Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.), Lecture Notes in Computer Science, Vol. 4111, Springer, 115–137.

[26] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Symbolic execution with separation logic. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS '05)*. Kwangkeun Yi (Ed.). Lecture Notes in Computer Science, Vol. 3780, Springer, 52–68.

[27] 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*.

[28] D. J. Bernstein. 2006. Curve25519: New Diffie-Hellman speed records. In *Proceedings of the IACR Conference on Practice and Theory of Public Key Cryptography (PKC)*.

[29] Daniel J. Bernstein. 2019. Public-Key Authenticated Encryption: Crypto_box. Retrieved from https://nacl.cr.yp.to/box.html

[30] Daniel J. Bernstein. 2005. Cache-Timing Attacks on AES. Retrieved from https://cr.yp.to/papers.html#cachetiming

[31] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales B. Paiva, Prasanna Ravi, and Goutam Tamvada. 2024. KyberSlash: Exploiting secret-dependent division timings in Kyber implementations. IACR Cryptology ePrint Archive 1049. Retrieved from https://eprint.iacr.org/2024/1049

[32] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. DY*: A modular symbolic verification framework for executable cryptographic protocol code. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P '21)*. IEEE, 523–542.

[33] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. An in-depth symbolic security analysis of the ACME standard. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery. DOI: https://doi.org/10.1145/3460120.3484588

[34] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. A tutorial-style introduction to DY*. In *Protocols, Logic, and Strands: Essays Dedicated to Joshua Guttman on the Occasion of His 66.66 Birthday*. Daniel Dougherty, José Meseguer, Sebastian Alexander Mödersheim, and Paul Rowe (Eds.), LNCS, Vol. 13066, Springer, 77–97.

[35] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017. Verified models and reference implementations for the TLS 1.3 standard candidate. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*, 483–502.

[36] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, K. Rustan M. Leino, Jay R. Lorch, et al. 2017. Everest: Towards a verified, drop-in

replacement of HTTPS. In *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL '17)*. Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), LIPIcs, Vol. 71, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 1, 1–12.

[37] Karthikeyan Bhargavan, Christina Brzuska, Cédric Fournet, Matthew Green, Markulf Kohlweiss, and Santiago Zanella-Béguelin. 2016. Downgrade resilience in key-exchange protocols. In *Proceedings of the IEEE Symposium on Security and Privacy (SP '16)*. IEEE Computer Society, 506–525.

[38] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. 2025. HAX: Verifying security-critical Rust software using multiple provers. In *Verified Software. Theories, Tools and Experiments*. Jonathan Protzenko, and Azalea Raad (Eds.), Springer Nature Switzerland, Cham, 96–119.

[39] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. 2013. Implementing TLS with verified cryptographic security. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 445–459.

[40] Karthikeyan Bhargavan, Lucas Franceschino, Franziskus Kiefer, and Goutam Tamvada. 2024. Verifying Libcrux's ML-KEM. Retrieved from https://cryspen.com/post/ml-kem-verification/

[41] Karthikeyan Bhargavan, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. 2025. Formal security and functional verification of cryptographic protocol implementations in Rust. Cryptology ePrint Archive, Paper 2025/980. Retrieved from https://eprint.iacr.org/2025/980

[42] Henk Birkholz, Christoph Vigano, and Carsten Bormann. 2019. Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures. RFC Editor. Retrieved from https://www.rfc-editor.org/info/rfc8610

[43] Bruno Blanchet. 2013. Automatic verification of security protocols in the symbolic model: The verifier ProVerif. In *Foundations of Security Analysis and Design VII—FOSAD 2012/2013 Tutorial Lectures*. Alessandro Aldini, Javier López, and Fabio Martinelli (Eds.), Lecture Notes in Computer Science, Vol. 8604, Springer, 54–87.

[44] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. 2011. Extending Sledgehammer with SMT solvers. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE '23)*. Nikolaj S. Bjørner and Viorica Sofronie-Stokkermans (Eds.), Lecture Notes in Computer Science, Vol. 6803, Springer, 116–130.

[45] Sascha Böhme and Tjark Weber. 2010. Fast LCF-style proof reconstruction for Z3. In *Proceedings of the 1st International Conference on Interactive Theorem Proving (ITP '10)*. Matt Kaufmann and Lawrence C. Paulson (Eds.), Lecture Notes in Computer Science, Vol. 6172, Springer, 179–194.

[46] 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*.

[47] Carsten Bormann and Paul E. Hoffman. 2020. Concise Binary Object Representation (CBOR). RFC Editor. Retrieved from https://www.rfc-editor.org/info/rfc8949

[48] 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)*.

[49] Jay Bosamiya, Wen Shih Lim, and Bryan Parno., August. 2022. Provably-safe multilingual software sandboxing using WebAssembly. In *Proceedings of the USENIX Security Symposium*.

[50] David Brumley and Dan Boneh., August. 2003. Remote timing attacks are practical. In *Proceedings of the USENIX Security Symposium*.

[51] ChrisBrzuska, Antoine Delignat-Lavaud, ChristophEgger, CédricFournet, Konrad Kohbrok and MarkulfKohlweiss. 2022. Key-schedule security for the TLS 1.3 standard. In *Advances in Cryptology – ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security*, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part I. Springer-Verlag, Berlin, 621–650. DOI: https://doi.org/10.1007/978-3-031-22963-3_21

[52] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. 2022. Security analysis of the MLS key derivation. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP)*. IEEE, 2535–2553.

[53] Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. 2022. Key-schedule security for the TLS 1.3 standard. In *Proceedings of the Advances in Cryptology: 28th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT '22), Part I*. Shweta Agrawal and Dongdai Lin (Eds.), Lecture Notes in Computer Science, Vol. 13791, Springer, 621–650.

[54] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. 2018. State separation for code-based game-playing proofs. In *Proceedings of the Advances in Cryptology: 24th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT '18), Part III*. Thomas Peyrin and Steven D. Galbraith (Eds.), Lecture Notes in Computer Science, Vol. 11274, Springer, 222–249.

[55] Chris Brzuska, Christoph Egger, and Jan Winkelmann. 2025. SSBee. Retrieved from https://github.com/sspverif/sspverif/

[56] Chris Brzuska and Sabine Oechsner. 2023. A state-separating proof for Yao's garbling scheme. In *Proceedings of the 36th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 137–152.

[57] Yi Cai, Pratap Singh, Zhengyao Lin, Jay Bosamiya, Joshua Gancher, Milijana Surbatovich, and Bryan Parno. 2025. Vest: Verified, secure, high-performance parsing and serialization for Rust. In *Proceedings of the USENIX Security Symposium*.

[58] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2016. Content security problems? Evaluating the effectiveness of content security policy in the wild. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, 1365–1375.

[59] Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu Lahiri, and Nikhil Swamy. 2025. Towards neural synthesis for SMT-assisted proof-oriented programming. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE '25)*. IEEE Press, 1755–1767.

[60] Saikat Chakraborty, Shuvendu Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-generated loop invariants for program verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. Houda Bouamor, Juan Pino, and Kalika Bali (Eds.), Association for Computational Linguistics, 9164–9175.

[61] Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. Type-preserving compilation of end-to-end verification of security enforcement. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*, ACM, New York, NY, 412–423.

[62] Xiangdong Chen, Zhaofeng Li, Lukas Mesicek, Vikram Narayanan, and Anton Burtsev. 2023. Atmosphere: Towards practical verified kernels in Rust. In *Proceedings of the Workshop on Kernel Isolation, Safety and Verification (KISV)*.

[63] 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)*.

[64] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. 2008. RFC 5280: Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile.

[65] James O. Coplien and Neil B. Harrison. 2004. *Organizational Patterns of Agile Software Development*. Prentice-Hall, Inc., Upper Saddle River, NJ.

[66] David Costanzo, Zhong Shao, and Ronghui Gu. 2016. End-to-end verification of information-flow security for C and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Chandra Krintz and Emery D. Berger (Eds.), ACM, 648–664.

[67] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.), ACM, 1773–1788.

[68] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. 2016. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *Proceedings of the IEEE Symposium on Security and Privacy (SP '16)*. IEEE Computer Society, 470–485.

[69] DARPA. 2024. Translating All C to Rust (TRACTOR). Retrieved September 2024 from https://www.darpa.mil/program/translating-all-c-to-rust

[70] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover. In *Proceedings of the of the Conference on Automated Deduction (CADE)*.

[71] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science, Vol. 4963, Springer, 337–340.

[72] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. 2017. Implementing and proving the TLS 1.3 record layer. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP '17)*. IEEE Computer Society, 463–482.

[73] Antoine Delignat-Lavaud, Cédric Fournet, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Jay Bosamiya, Joseph Lallemand, Itsaka Rakotonirina, and Yi Zhou. 2021. A security model and fully verified implementation for the IETF QUIC record layer. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (SP '21)*. IEEE, 1162–1178.

[74] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A foundry for the deductive verification of Rust programs. In *Proceedings of the International Conference on Formal Engineering Methods*.

[75] 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)*.

[76] Jason A. Donenfeld. 2017. Wireguard: Next Generation Kernel Network Tunnel. Retrieved January 2017 from https://www.wireguard.com/

[77] Jason A. Donenfeld. 2018. New 25519 Measurements of Formally Verified Implementations. Retrieved February 2018 from http://moderncrypto.org/mail-archive/curves/2018/000972.html

[78] Nir Drucker and Shay Gueron. 2021. Selfie: Reflections on TLS 1.3 with PSK. *Journal of Cryptology* 34, 3 (2021), 27.

[79] François Dupressoir, Konrad Kohbrok, and Sabine Oechsner. 2022. Bringing state-separating proofs to EasyCrypt a security proof for cryptobox. In *Proceedings of the 35th IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 227–242.

[80] Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 2025. PulseCore: An impredicative concurrent separation logic for dependently typed programs. In *Proceedings of the 46th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '25)*. ACM.

[81] Christoph Egger. 2023. On Abstraction and Modularization in Protocol Analysis. Doctoral thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU).

[82] Marco Eilers, Malte Schwerhoff, Alexander J. Summers, and Peter Müller. 2025. Fifteen years of Viper. In *Proceedings of the 46th 37th International Conference on Computer Aided Verification (CAV '25), Part I*. Ruzica Piskac and Zvonimir Rakamaric (Eds.), Lecture Notes in Computer Science, Vol. 15931, Springer, 107–123.

[83] Burak Ekici, Guy Katz, Chantal Keller, Alain Mebsout, Andrew J. Reynolds, and Cesare Tinelli. 2016. Extending SMTCoq, a certified checker for SMT (extended abstract). In *Proceedings 1st International Workshop on Hammers for Type Theories (HaTT@IJCAR '16)*. Jasmin Christian Blanchette and Cezary Kaliszyk (Eds.), EPTCS, Vol. 210, 21–29.

[84] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 1889–1912.

[85] 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*.

[86] Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Samuel Gruetter, Clément Pit-Claudel, and Adam Chlipala. 2024. Foundational integration verification of a cryptographic server. *Proceedings of the ACM on Programming Languages* 8, PLDI (June 2024), 1704–1729.

[87] Sarah Fakhoury, Markus Kuppe, Shuvendu K. Lahiri, Tahina Ramananandro, and Nikhil Swamy. 2025. 3DGen: AI-assisted generation of provably correct binary format parsers. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE '25)*. IEEE Press, 2535–2547.

[88] N. J. Al Fardan and K. G. Paterson. 2013. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[89] 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)*.

[90] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1229–1241.

[91] Michael Fitzgibbons, Zoe Paraskevopoulou, Noble Mushtak, Michelle Thalakottur, Jose Sulaiman Manzur, and Amal Ahmed. 2024. RichWasm: Bringing safe, fine-grained, shared-memory interoperability down to WebAssembly. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1656–1679.

[92] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. 2011. Modular code-based cryptographic verification. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, New York, NY, 341–350.

[93] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. 2019. A verified, efficient embedding of a verifiable assembly language. *Proceedings of the ACM on Programming Languages* 3, POPL (January 2019), 1–30.

[94] Aymeric Fromherz and Jonathan Protzenko. 2024. Compiling C to safe Rust, formalized. arXiv:2412.15042. Retrieved from https://arxiv.org/abs/2412.15042

[95] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: Proof-oriented programming in a dependently typed concurrent separation logic. *Proceedings of the ACM on Programming Languages* 5, ICFP (August 2021), 1–30.

[96] Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, Computers, Controls* 2, 5 (1971), 45–50.

[97] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2024. RefinedRust: A type system for high-assurance verification of Rust. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1115–1139.

[98] Joshua Gancher, Sydney Gibson, Pratap Singh, Samvid Dharanikota, and Bryan Parno. 2023. Owl: Compositional verification of security protocols via an information-flow type system. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[99] Google. 2022. Announcing KataOS and Sparrow. Retrieved October 2022 from https://opensource.googleblog.com/2022/10/announcing-kataos-and-sparrow.html

[100] Niklas Grimm, Kenji Maillard, Cédric Fournet, Cătălin Hriţcu, Matteo Maffei, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, and Santiago Zanella-Béguelin. 2018. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '18)*. ACM, New York, NY, 130–145.

[101] Samuel Gruetter, Viktor Fukala, and Adam Chlipala. 2024. Live verification in an interactive proof assistant. *Proceedings of the ACM on Programming Languages* 8, PLDI (June 2024), 1535–1558.

[102] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, 653–669.

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

[104] Sean Gulley, Vinodh Gopal, Kirk Yap, Wajdi Feghali, Jim Guilford, and Gil Wolrich. 2013. Intel® SHA Extensions. Retrieved July 2013 from https://software.intel.com/sites/default/files/article/402097/intel-sha-extensions-white-paper.pdf

[105] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. *ACM SIGPLAN Notices* 52, 6 (June 2017), 185–200.

[106] Philipp G. Haselwarter, Benjamin Salling Hvass, Lasse Letager Hansen, Théo Winterhalter, Cătălin Hriţcu, and Bas Spitters. 2024. The last yard: Foundational end-to-end verification of high-speed cryptography. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '24)*, ACM, New York, NY, 30–44.

[107] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Cătălin Hriţcu, Kenji Maillard, and Bas Spitters. 2023. SSProve: A foundational framework for modular cryptographic proofs in Coq. *ACM Transactions on Programming Languages and Systems* 45, 3, Article 15 (2023), 1–61.

[108] 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)*.

[109] 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 Systems Design and Implementation (OSDI)*.

[110] Son Ho, Aymeric Fromherz, and Jonathan Protzenko. 2023. Modularity, code specialization, and zero-cost abstractions for program verification. *Proceedings of the ACM on Programming Languages* 7, ICFP (August 2023), 385–416.

[111] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages 6, ICFP (*2022), 711–741.

[112] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. 2022. Noise*: A library of verified high-performance secure channel protocol implementations. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP '22)*. IEEE, 107–124.

[113] Jana Hofmann, Cédric Fournet, Boris Köpf, and Stavros Volos. 2024. Gaussian elimination of side-channels: Linear algebra for memory coloring. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. ACM, New York, NY, 2799–2813.

[114] A. Hunt and D. Thomas. 2002. Software archaeology. *IEEE Software* 19, 2 (2002), 20–22.

[115] Chung-Kil Hur and Derek Dreyer. 2011. A Kripke logical relation between ML and assembly. *ACM SIGPLAN Notices* 46, 1 (January 2011), 133–146.

[116] Graham Hutton. 1989. Parsing using combinators. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. Springer-Verlag, Berlin, 353–370.

[117] Intel. 2014. New instructions supporting large integer arithmetic on Intel Architecture processors. Retrieved from https://raw.githubusercontent.com/wiki/intel/intel-ipsec-mb/doc/ia-large-integer-arithmetic-paper.pdf

[118] ITU Recommendation X 680. 2021. ITU-T study group 17. X.680: Information technology—Abstract syntax notation one (ASN.1): Specification of basic notation. Retrieved from https://standards.globalspec.com/std/14381752/x-680

[119] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of the 3rd International Symposium on NASA Formal Methods (NFM '11)*. Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.), Lecture Notes in Computer Science, Vol. 6617, Springer, 41–55.

[120] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, 275–288.

[121] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.

[122] Ben Kallus, Prashant Anantharaman, Michael Locasto, and Sean W. Smith. 2024. The HTTP garden: Discovering parsing vulnerabilities in HTTP/1.1 implementations by differential fuzzing of request streams.

[123] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding inductive loop invariants using large language models. arXiv:2311.07948. Retrieved from https://arxiv.org/abs/2311.07948

[124] Ioannis T. Kassios. 2006. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Proceedings of the 14th International Conference on Formal Methods (FM '06)*. Springer-Verlag, Berlin, 268–283.

[125] Franziskus Kiefer and Karthikeyan Bhargavan. 2024. Formally Verified Post-Quantum Cryptography. Retrieved August 2024 from https://cryspen.com/post/fospqc/

[126] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems* 32, 1, Article 2 (February 2014), 1–70.

[127] Konrad Kobrok, Markulf Kohlweiss, Tahina Ramananandro, and Nikhil Swamy. 2020. Relational F* for State Separating Cryptographic Proofs. Retrieved January 2020 from https://github.com/FStarLang/FStar/wiki/Relational-F*-for-State-Separating-Cryptographic-Proofs

[128] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP)*, 1–19.

[129] Paul C. Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings of the International Cryptology Conference (CRYPTO)*.

[130] Konrad Kohbrok. 2023. State-Separating Proofs and Their Applications. Doctoral thesis, Aalto University School of Science.

[131] Matthew Kolosick, Basavesh Ammanaghatta Shivakumar, Sunjay Cauligi, Marco Patrignani, Marco Vassena, Ranjit Jhala, and Deian Stefan. 2025. Robust constant-time cryptography. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*.

[132] Shuvendu K. Lahiri. 2024. AI-assisted user intent formalization for programs: Problem and applications (invited talk). In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware '24)*. Bram Adams, Thomas Zimmermann, Ipek Ozkaya, Dayi Lin, and Jie M. Zhang (Eds.), ACM.

[133] Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. 2022. Hypertree proof search for neural theorem proving. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 35, 26337–26349.

[134] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133–169.

[135] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, and J. Iyengar. 2017. The QUIC transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 183–196.

[136] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, et al. 2024. Verus: A practical foundation for systems verification. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[137] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust programs using linear ghost types. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[138] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid types for Rust. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 1533–1557.

[139] K. R. M. Leino and Clément Pit-Claudel. 2016. Trigger selection strategies to stabilize program verifiers. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. Swarat Chaudhuri and Azadeh Farzan (Eds.).

[140] K. Rustan M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '10)*. Springer-Verlag, Berlin, 348–370.

[141] K. Rustan, M. Leino, and Nadia Polikarpova. 2014. Verified calculations. In *Proceedings of the Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*.

[142] Xavier Leroy. 2006. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In *Proceedings of 33rd Symposium Principles of Programming Languages (POPL '06)*. ACM, 42–54.
[143] Pierre Letouzey. 2008. Extraction in Coq: An overview. In *Proceedings of the 4th Conference on Computability in Europe: Logic and Theory of Algorithms (CiE '08)*, Springer-Verlag, Berlin, 359–369.
[144] Kirby Linvill, Gowtham Kaki, and Eric Wustrow. 2023. Verifying indistinguishability of privacy-preserving protocols. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (October 2023), 1442–1469.
[145] Chang Liu, Xiwei Wu, Yuan Feng, Qinxiang Cao, and Junchi Yan. 2023. Towards general loop invariant generation via coordinating symbolic execution and large language models. arXiv:2311.10483. Retrieved from https://arxiv.org/abs/2311.10483
[146] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Computing Surveys* 54, 6 (July 2021), 1–37.
[147] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra monads for all. *Proceedings of the ACM on Programming Languages* 3, ICFP (July 2019), 1–29.
[148] Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The next 700 relational program logics. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 4 (2020), 1–33.
[149] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, et al. 2019. Meta-F*: Proof automation with SMT, tactics, and metaprograms. In *Proceedings of the 28th European Symposium on Programming Languages and Systems (ESOP '19), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS '19)*. Luís Caires (Ed.), Lecture Notes in Computer Science, Vol. 11423, Springer, 30–59.
[150] Lúcás C. Meier. 2022. State Separable Proofs for the Curious Cryptographer. Blogpost. Retrieved from https://cronokirby.com/posts/2022/05/state-separable-proofs-for-the-curious-cryptographer/
[151] Microsoft. 2023. Rust for Windows, and the Windows Crate. Retrieved August 2023 from https://learn.microsoft.com/en-us/windows/dev-environment/rust/rust-for-windows
[152] Shane Miller and Carl Lerche. 2022. Sustainability with Rust. Retrieved February 2022 from https://aws.amazon.com/blogs/opensource/sustainability-with-rust/
[153] Mozilla. 2018. Measurement Dashboard. Retrieved July 2018 from https://mzl.la/2ug9YCH
[154] Mozilla. 2021. Mozilla Welcomes the Rust Foundation. Retrieved February 2021 from https://blog.mozilla.org/en/mozilla/mozilla-welcomes-the-rust-foundation/
[155] National Institute of Standards and Technology. 2012. Secure Hash Standard (SHS). FIPS PUB 180–4.
[156] Haobin Ni, Antoine Delignat-Lavaud, Cédric Fournet, Tahina Ramananandro, and Nikhil Swamy. 2023. ASN1*: Provably correct, non-malleable parsing for ASN.1 DER. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23)*. Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic (Eds.), ACM, 275–289.
[157] NIST. 2001. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*. NIST Special Publication 800-38A.
[158] NIST. 2001. *Announcing the Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197.
[159] NIST. 2007. *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. NIST Special Publication 800-38D.
[160] Peter W. O'Hearn. 2004. Resources, concurrency and local reasoning. In *CONCUR 2004—Concurrency Theory*, Philippa Gardner and Nobuko Yoshida (Eds.), Springer, Berlin, 49–67.
[161] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing black-box CPUs against speculation contracts. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*, ACM, New York, NY, 226–239.
[162] Thomaz Oliveira, Julio López, Hüseyin Hışıl, Armando Faz-Hernández, and Francisco Rodríguez-Henríquez. 2017. How to (pre-)compute a ladder: Improving the performance of X25519 and X448. In *Proceedings of Selected Areas in Cryptography (SAC)*.
[163] Daniel Patterson, Noble Mushtak, Andrew Wagner, and Amal Ahmed. 2022. Semantic soundness for language interoperability. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*. Ranjit Jhala and Isil Dillig (Eds.), ACM, 609–624.
[164] Daniel Patterson, Jamie Perconti, Christos Dimoulas, and Amal Ahmed. 2017. FunTAL: Reasonably mixing a functional language with assembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. Albert Cohen and Martin T. Vechev (Eds.), ACM, 495–509.
[165] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the keyboard? Assessing the security of GitHub copilot's code contributions. *Communications of the ACM* 68, 2 (January 2025), 96–105.

[166] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants? In *Proceedings of the 40th International Conference on Machine Learning (ICML '23)*. JMLR.org.

[167] Colin Percival. 2005. Cache Missing for Fun and Profit. Retrieved from https://papers.freebsd.org/2005/cperciva-cache_missing/

[168] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella Béguelin. 2020. HACLxN: Verified generic SIMD crypto (for all your favourite platforms). In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.), ACM, 899–918.

[169] 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)*.

[170] François Pottier. 2017. Visitors unchained. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–28.

[171] Rachel Potvin and Josh Levenberg. 2016. Why Google stores billions of lines of code in a single repository. *Communications of the ACM* 59, 7 (June 2016), 78–87.

[172] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. 2020. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[173] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, et al. 2017. Verified low-level programming embedded in F*. *Proceedings of the ACM on Programming Languages* 1, ICFP (August 2017), 1–29.

[174] Proven Liam. 2022. Linux 6.1: Rust to Hit Mainline Kernel. Retrieved October 2022 from https://www.theregister.com/2022/10/05/rust_kernel_pull_request_pulled/

[175] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. 2024. Towards AI-assisted synthesis of verified Dafny methods. arXiv:2402.00247. Retrieved from https://arxiv.org/abs/2402.00247

[176] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified secure zero-copy parsers for authenticated message formats. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*. Nadia Heninger and Patrick Traynor (Eds.), USENIX Association, 1465–1482.

[177] Tahina Ramananandro, Gabriel Ebner, Guido Martínez, and Nikhil Swamy. 2025. Secure Parsing and Serializing with Separation Logic, Applied to CBOR, CDDL, and COSE. Retrieved May 2025 from https://arxiv.org/abs/2505.17335

[178] Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2021. EverParse: Hardening Critical Attack Surfaces with Formally Proven Message Parsers. Retrieved May 2021 from https://www.microsoft.com/en-us/research/blog/everparse-hardening-critical-attack-surfaces-with-formally-proven-message-parsers/

[179] Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. 2020. Programming and Proving with Indexed Effects. Retrieved July 2020 from https://fstar-lang.org/papers/indexedeffects/indexedeffects.pdf

[180] Antonin Reitz, Aymeric Fromherz, and Jonathan Protzenko. 2024. StarMalloc: Verifying a modern, hardened memory allocator. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (October 2024), 1757–1786.

[181] Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, et al. 2025. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. arXiv:2504.21801. Retrieved from https://arxiv.org/abs/2504.21801

[182] E. Rescorla. 2018. The transport layer security (TLS) protocol version 1.3. Retrieved from https://www.rfc-editor.org/info/rfc8446

[183] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. 2025. TLS Encrypted Client Hello. RFC Editor. Retrieved from https://www.rfc-editor.org/info/rfc9849

[184] John C. Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, Washington, DC, *55–74*.

[185] Talia Ringer. 2021. Proof Repair. Ph.D. thesis, University of Washington, USA.

[186] Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. 2020. REPLica: REPL instrumentation for Coq analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20)*. ACM, New York, NY, 99–113.

[187] Amos Robinson and Alex Potanin. 2024. Pipit on the post: Proving pre- and post-conditions of reactive systems. In *Proceedings of the 38th European Conference on Object-Oriented Programming (ECOOP '24)*. Jonathan Aldrich and Guido Salvaneschi (Eds.), LIPIcs, Vol. 313, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Article 34, 1–28.

[188] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. 2020. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 1–10.

[189] Jim Schaad. 2022. CBOR Object Signing and Encryption (COSE): Structures and Process. RFC 9052. Retrieved from https://www.rfc-editor.org/info/rfc9052

[190] Sheera Shamsu, Dipesh Kafle, Dhruv Maroo, Kartik Nagar, Karthikeyan Bhargavan, and K. C. Sivaramakrishnan. 2025. A mechanically verified garbage collector for OCaml. *Journal of Automated Reasoning* 69, 2 (2025), 11.

[191] Lucas Silver and Steve Zdancewic. 2021. Dijkstra monads forever: Termination-sensitive specifications for interaction trees. *Proceedings of the ACM on Programming Languages* 5, POPL (January 2021), 1–28.

[192] Pratap Singh, Joshua Gancher, and Bryan Parno. 2025. OwlC: Compiling security protocols to verified, secure, high-performance libraries. In *Proceedings of the USENIX Security Symposium*.

[193] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2025. Correct and complete type checking and certified erasure for Coq, in Coq. *Journal of the ACM* 72, 1 (November 2025), 1–74.

[194] Pierre-Yves Strub, Nikhil Swamy, Cédric Fournet, and Juan Chen. 2012. Self-certification: Bootstrapping certified typecheckers in F* with Coq. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*. ACM, 571–584.

[195] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2023. Clover: Closed-loop verifiable code generation. arXiv:2310.17807. Retrieved from https://arxiv.org/abs/2310.17807

[196] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, et al. 2024. Anvil: Verifying liveness of cluster management controllers. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[197] Nikhil Swamy, Juan Chen, and Ravi Chugh. 2010. Enforcing stateful authorization and information flow policies in fine. In *Proceedings of the European Symposium on Programming (ESOP)*.

[198] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. *ACM SIGPLAN Notices* 46, 9 (September 2011), 266–278.

[199] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2006. Safe manual memory management in cyclone. *Science of Computer Programming* 62, 2 (October 2006), 122–144.

[200] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. 2016. Dependent types and multi-monadic effects in F*. *ACM SIGPLAN Notices* 51, 1 (January 2016), 256–270.

[201] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. 2022. Hardening attack surfaces with formally proven binary format parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*. Ranjit Jhala and Isil Dillig (Eds.), ACM, 31–45.

[202] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: An extensible concurrent separation logic for effectful dependently typed programs. *Proceedings of the ACM on Programming Languages* 4, ICFP (August 2020), 1–30.

[203] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, 387–398.

[204] Zhe Tao, Aseem Rastogi, Naman Gupta, Kapil Vaswani, and Aditya V. Thakur., August. 2021. DICE*: A formally verified implementation of DICE measured boot. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*. USENIX Association, 1091–1107.

[205] Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. 2023. A language-agent approach to formal theorem-proving. arXiv:2310.04353. Retrieved from https://arxiv.org/abs/2310.04353

[206] Aaron Tomb. 2016. Automated verification of real-world cryptographic implementations. *IEEE Security Privacy Magazine* 14, 6 (November. 2016), 26–33.

[207] Trusted Computing Group. 2023. DICE Protection Environment, Version 1.0, Revision 0.6. Retrieved from https://trustedcomputinggroup.org/wp-content/uploads/TCG-DICE-Protection-Environment-Specification_14february2023-1.pdf

[208] 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)*.

[209] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying dynamic trait objects in Rust. In *Proceedings of the IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[210] Niki Vazou and Michael Greenberg. 2022. How to safely use extensionality in Liquid Haskell. arXiv:2103.02177. Retrieved from https://arxiv.org/abs/2103.02177

[211] Stavros Volos, Cédric Fournet, Jana Hofmann, Boris Köpf, and Oleksii Oleksenko. 2024. Principled microarchitectural isolation on cloud CPUs. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.), ACM, 183–197.

[212] P. Wadler and S. Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, 60–76.

[213] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. 2023. TreeSync: Authenticated group management for messaging layer security. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23)*, 1217–1233.

[214] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. 2023. Comparse: Provably secure formats for cryptographic protocols. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. ACM, New York, NY, 564–578.

[215] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. 2025. TreeKEM: A modular machine-checked symbolic security analysis of group key agreement in messaging layer security. IACR Cryptology ePrint Archive, 410.

[216] Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. 2025. Kimina-Prover Preview: Towards Large Formal Reasoning Models with Reinforcement Learning. arXiv: 2504.11354. Retrieved from https://arxiv.org/abs/2504.11354

[217] White House Office of the National Cyber Director. 2024. Back to the Building Blocks: A Path toward Secure and Measurable Software. Retrieved February 2024 from https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[218] Théo Winterhalter, Cezar-Constantin Andrici, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, and Exequiel Rivas. 2022. Partial Dijkstra monads for all. In *Proceedings of the 28th International Conference on Types for Proofs and Programs (TYPES)*.

[219] Fabian Wolff, Aurel Bílý, Christoph Matheja, Peter Müller, and Alexander J. Summers. 2021. Modular specification and verification of closures in Rust. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (October 2021), 1–29.

[220] Jenny Xiang, Daniel Schoepe, and Marianna Rapoport. 2025. Transitioning production Software Verification to Verus: An Experience Report. Retrieved May 2025 from https://sites.google.com/view/rustverify2025

[221] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *Proceedings of the International Conference on Machine Learning*. PMLR, 6984–6994.

[222] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. 2024. LeanDojo: Theorem proving with retrieval-augmented language models. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 36.

[223] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2010. CacheBleed: A timing attack on OpenSSL constant time RSA. In *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems (CHES)*.

[224] 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)*.

[225] Miao Yu, Virgil Gligor, and Limin Jia. 2021. An I/O separation model for formal verification of kernel implementations. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[226] Yi Zhou, Jay Bosamiya, Jessica Li, Marijn Heule, and Bryan Parno. 2024. Context pruning for more robust SMT-based program verification. In *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD) Conference*.

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

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

[229] Yi Zhou, Amar Shah, Marijn Heule, and Bryan Parno. 2025. Cazamariposas: Automated instability debugging in SMT-based program verification. In *Proceedings of the Conference on Automated Deduction (CADE)*.

[230] Ziqiao Zhou, Weiteng Chen, Chris Hawblitzel, and Weidong Cui. 2024. VeriSMo: A verified security module for confidential VMs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[231] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, 1789–1806.

[232] Sacha Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 2024. A hybrid approach to semi-automated Rust verification. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM.

Appendix

A Contributors to Project Everest

We list below, in alphabetical order, people who co-authored at least one peer-reviewed paper that was published by the Project Everest team, as well as some engineers who helped build core parts of our infrastructure.

Carmine Abate, Danel Ahman, Robert Atkey, Evmorfia-Iro Bartzia, Benjamin Beurdouche, Karthikeyan Bhargavan, Barry Bond, Jay Bosamiya, Chris Brzuska, Omar Cardona, Tej Chajed, Joonwon Choi, Eric Cornelissen, Antoine Delignat-Lavaud, Victor Dumitrescu, Christoph Egger, Cédric Fournet, Aymeric Fromherz, Nick Giannarakis, Sydney Gibson, Niklas Grimm, Arti Gupta, Philipp G. Haselwarter, Chris Hawblitzel, Cătălin Hriţcu, Samin Ishtiaq, Srikanth Kannepalli, Manos Kapritsos, Nadim Kobeissi, Konrad Kohbrok, Markulf Kohlweiss, Natalia Kulatova, Joseph Lallemand, K. Rustan M. Leino, Yao Li, Jacob R. Lorch, Matteo Maffei, Kenji Maillard, Dmitry Malloy, Guido Martínez, Denis Merigoux, Monal Narasimhamurthy, Haobin Ni, Jianyang Pan, Zoe Paraskevopoulou, Bryan Parno, Clément Pit-Claudel, Gordon Plotkin, Marina Polubelova, Jonathan Protzenko, Itsaka Rakotonirina, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Exequiel Rivas, Srinath Setty, Irina Spiridonova, Bas Spitters, Nikhil Swamy, Michael Tang, Éric Tanter, Laure Thompson, Antoine Van Muylder, Gustavo Varo, Juan Vazquez, Peng Wang, Théo Winterhalter, Christoph M. Wintersteiger, Santiago Zanella-Béguelin, Yi Zhou, and Jean Karim Zinzindohoue.

We are also very grateful to the many open source contributors to all of our projects, and others who have continued to build on and improve the artifacts that Project Everest created.

Received 17 June 2025; revised 25 September 2025; accepted 17 November 2025