Everest: Towards a Verified, Drop-in Replacement of HTTPS

Karthikeyan Bhargavan[1], Barry Bond[2], Antoine Delignat-Lavaud[2], Cédric Fournet[2], Chris Hawblitzel[2], Cătălin Hriţcu[1], Samin Ishtiaq[2], Markulf Kohlweiss[2], Rustan Leino[2], Jay Lorch[2], Kenji Maillard[1], Jianyang Pang[1], Bryan Parno[2], Jonathan Protzenko[2], Tahina Ramananandro[2], Ashay Rane[2], Aseem Rastogi[2], Nikhil Swamy[2], Laure Thompson[2], Perry Wang[2], Santiago Zanella-Béguelin[2], and Jean-Karim Zinzindohoué[1]

1 **INRIA Paris**
2 **Microsoft Research**

Attendance statement: The work will be presented by Nikhil Swamy.

Justification statement: Our submission presents *Everest*, a 5-year joint project between Microsoft Research and INRIA, begun last year. We aim to build a verified replacement of the HTTPS stack. In contrast with other large-scale verification efforts, we aim to deploy *Everest* software within the existing software ecosystem. As such, we envisage balancing a shift towards verified software development with an incremental approach to the deployment of verified software. This raises a number of open issues, e.g., how to explain the benefits of formal verification to the software industry at large; how will verified code and proofs be maintained and evolved once deployed; etc. We hope our presentation will spur some discussions on these topics.

The main technical content of the paper sketches our verification methodology, which involves the use of advanced programming language features, especially dependently typed (meta-)programming and proving, to carry out high-level proofs of correctness and security for low-level, efficient cryptographic implementations. We summarize our main results so far, drawing from other, more detailed, formal presentations of our work.

Abstract

The HTTPS ecosystem is the foundation on which Internet security is built. At the heart of this ecosystem is the Transport Layer Security (TLS) protocol, which in turn uses the X.509 public-key infrastructure and numerous cryptographic constructions and algorithms. Unfortunately, this ecosystem is extremely brittle, with headline-grabbing attacks and emergency patches many times a year. We describe our ongoing efforts in *Everest*,[1] a project that aims to build and deploy a verified version of TLS and other components of HTTPS, replacing our infrastructure with proven, secure software.

Aiming both at full verification and usability, we conduct high-level code-based, game-playing proofs of security on cryptographic implementations that yield efficient, deployable code, at the level of C and assembly. Concretely, we use F*, a dependently typed language for (meta-)programming and proving at a high level, while relying on low-level DSLs embedded within F* for programming low-level components when necessary for performance and, sometimes, side-channel resistance. To compose the pieces, we compile all our code to source-like C and assembly, suitable for audit and deployment by independent security experts.

Our main results so far include (1) the design of Low*, a subset of F* designed for C-like imperative programming but with high-level verification support, and KreMLin, a compiler

[1] The *Everest* VERified End-to-end Secure Transport: https://project-everest.github.io

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

that extracts Low* programs to C; (2) an implementation of the TLS-1.3 record layer in Low*, together with a proof of its concrete cryptographic security; (3) Vale, a new DSL for verified assembly language, and several optimized cryptographic algorithms proven functionally correct. In an early deployment, all our verified software is integrated and deployed within `libcurl`, a widely used library of networking protocols.

1 Introduction

The Internet's core security infrastructure is frighteningly brittle. As more and more services rely on encryption, security best practices urge developers to use standard, widely-used components like HTTPS and SSL (the latter is now standardized as TLS). As a result, the same pervasive components are used for securing communications on the Web and for VoIP, email, VPNs, and the IoT.

Unfortunately, these standard components are themselves often broken in many ways. Even before recent headline-grabbing attacks like HeartBleed, FREAK, and Logjam, entire papers were published just to summarize all of the academically "interesting" ways TLS implementations have been broken, without even getting into "boring" vulnerabilities like buffer overflows and other basic coding mistakes. This tide of flaws shows no signs of abating; in the year after those papers were published, 54 new CVE security advisories were published just for TLS. These flaws are frequently found and fixed in all of the widely used TLS implementations, as well as in the larger HTTPS ecosystem. They span a wide gamut including memory management mistakes, errors in protocol state machines, lax X.509 certificate parsing and validation, weak or badly implemented cryptographic algorithms, side channels, and even genuine design flaws in the standards. Furthermore, because many TLS implementations expose truly terrible APIs, HTTPS applications built on them regularly make devastating mistakes.

These persistent problems have generated sufficient industry concern that both Google and the OpenBSD project are building separate forks of OpenSSL (BoringSSL and LibreSSL, respectively) while Amazon is developing a brand new implementation. Many corporations has even joined the multi-million-dollar Core Infrastructure Initiative to support additional testing and security auditing of open-source projects, starting with OpenSSL.

1.1 A Need for Verified Deployments Now

While the industry is taking incremental steps to try to stem the persistent tide of vulnerabilities, the programming-language community is uniquely positioned to definitively solve this problem. The science of software verification has progressed to a point where a large team of experts can reasonably expect to build a fully verified software stack, e.g., SEL4 [8], Ironclad [6], and CertiKOS [5], with still more ambitious, broadly ranging efforts already underway (e.g., `http://deepspec.org/`). Yet, even when augmented with secure communication components like TLS, a fully verified stack would not meet today's pressing needs, since a wholesale replacement of the software stack is not in the offing.

Improving the current software landscape requires both a dramatic shift in software development methodology and an incremental approach to the deployment of verified software. *Everest* is a new joint project between Microsoft Research and INRIA which aims to build verified software components and deploy them within the existing software ecosystem. Specifically, *Everest* develops new implementations of existing protocols and standards used for web communications. At a minimum, we prove our implementations functionally correct. Beyond functional correctness, we integrate cryptographic modeling and protocol analysis

to prove, by reduction to computational assumptions on their core algorithms, that our implementations provide secure-channel abstractions between the communicating endpoints. Our verified code is implemented in F⋆ [15], a dependently typed programming language and proof assistant, and in several DSLs embedded within F⋆.

After verification, in support of incremental deployment, our code is extracted by verified tools to C and assembly, and compiled further by off-the-shelf C compilers (e.g., gcc and clang, but also, at a performance cost, verified compilers like CompCert [10]) and composed with adapters that interface our verified code with existing software components, like the web browsers, servers and other communication software shown in the Figure 1. Being only as strong as its weakest component, software systems that includes verified *Everest* code may not be impervious to attack; yet, any attack such a system suffers will be attributable to a flaw in a component outside *Everest*, while simple, critical systems may be within reach of full verification with a reasonable marginal effort.

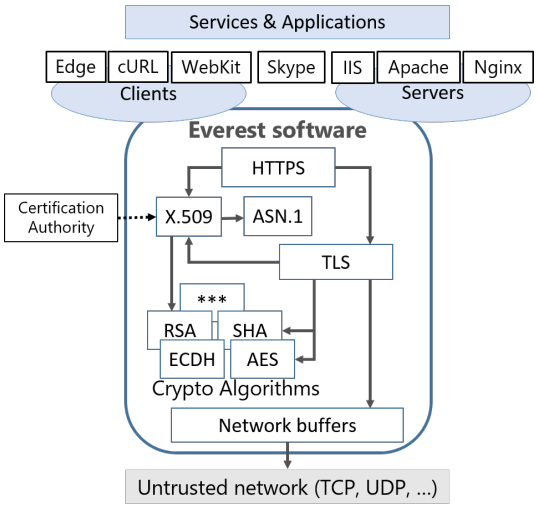

Figure 1 *Everest* architecture

1.2 Structure of this paper

We present an overview of the methodology
we have used so far in *Everest*. Our main guiding principle is to provide low-level, efficient implementations of various protocol standards by extracting them from fresh code, programmed and verified at a high-level of abstraction. This principle applies best for relatively small and complex code, such as a secure networking stack.

To this end, in §2, we present Low⋆, an embedded sub-language of F⋆ geared towards programming against a C-like memory model, while specifying and proving these programs within F⋆'s dependent type theory. Low⋆ programs are extracted to C by a new tool called KreMLin, which we have formally modeled as soundly preserving the functionality of programs to the level of CompCert's Clight. We also prove that compilation from Low⋆ to Clight does not introduce any side-channels due to memory access patterns.

In §3, we sketch several examples of verified code and their specifications in Low⋆, showing how we state and prove memory safety, functional correctness, and cryptographic security.

In §4, we discuss a few strands of ongoing work. This includes the design of a new DSL for verified assembly language programming and its use in producing even lower level, efficient, functionally correct implementations of the AES and SHA256 algorithms whose performance is on par with OpenSSL, the mostly widely used implementation. We also discuss an early deployment of *Everest* software as a drop-in replacement for TLS in libcurl, and its use from within a command line git client.

Finally, §5 presents some parting thoughts, covering some opportunities and challenges.

2 Low⋆: Verified Low-level Programming Embedded in F⋆

We aim to bridge the gap between high-level, safe-by-construction code, optimized for clarity and ease of verification, and low-level code exerting fine control over data representations and memory layout in order to achieve better performance. Towards this end, we introduce Low⋆, a DSL for verified, efficient low-level programming, embedded within F⋆, a verification-oriented, ML-like language with dependent types. Figure 2 illustrates the high-level design of Low⋆ and its compilation to native code.

Libraries for low-level programming within F⋆ At its core, F⋆ is a purely functional language to which effects like state are added programmatically using monads. We instantiate the state monad of F⋆ to use a CompCert-like structured memory model [10, 11] that separates the stack and the heap, supporting transient allocation on the stack, and allocating and freeing individual reference cells on the heap—this is not the "big array of bytes" model systems programmers sometimes use. The heap is organized into disjoint logical regions, which enables stating separation properties necessary for modular, stateful verification. On top of this, we program a library of buffers—C-style arrays passed by reference—with support for pointer arithmetic and referring to only part of an array. By virtue of F⋆ typing, our libraries and all their well-typed clients are guaranteed to be memory safe, e.g., they never access out-of-bounds or deallocated memory.

Designing Low⋆, a subset of F⋆ easily compiled to C We intend to give Low⋆ programmers precise control over the performance profile of the generated C code. Inasmuch as possible, we aim for the programmer to have control even over the syntactic structure of the target C code, to facilitate its review by security experts unfamiliar with F⋆. As such, to a first approximation, Low⋆ programs are F⋆ programs well-typed in the state monad described above, which, after all their computationally irrelevant (ghost) parts have been erased, must satisfy several requirements. Specifically, the code (1) must be first order, to prevent the need to allocate closures in C; (2) must not perform any implicit allocations; (3) must not use recursive datatypes, since these would have to be compiled using additional indirections to C structs; and (4) must be monomorphic, since C does not support polymorphism directly. We emphasize that these restrictions apply only to computationally relevant code—proofs and specifications are free to use arbitrary higher-order, dependently typed F⋆.

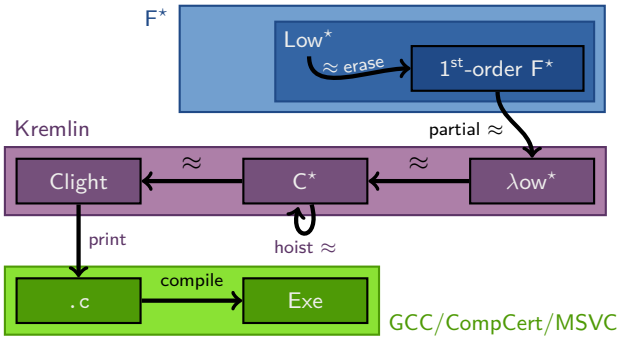

Figure 2 Low⋆ embedded in F⋆, compiled to C, with soundness and security guarantees

A dual interpretation of Low⋆, via compilation to OCaml or Clight Low⋆ programs interoperate naturally with other F⋆ programs, and precise specifications of Low⋆ and F⋆ code are intermingled when proving properties of their combination. As usual in F⋆, programs are type-checked and compiled to OCaml for execution, after erasing computationally irrelevant parts of a program, e.g., proofs and specifications, using a process similar to Coq's extraction mechanism [12]. Importantly, Low⋆ programs have a second, equivalent but more efficient semantics via compilation to C, with a predictable performance model including manual

memory management—this compilation is implemented by KreMLin, a new compiler from the Low⋆ subset of F⋆ to C.

Justifying its dual interpretation as a subset of F⋆ and a subset of C, we give a translation from Low⋆, via two intermediate languages, to CompCert's Clight [4] and show that it preserves trace equivalence with respect to the original F⋆ semantics of the program. In addition to ensuring that the functional behavior of a program is preserved, our trace equivalence also guarantees that the compiler does not introduce unexpected side-channels due to memory access patterns, at least until it reaches Clight—a useful sanity check for cryptographic code.

KreMLin, a compiler from Low⋆ to C Our formal model guides the implementation of KreMLin, a new tool that emits C code from Low⋆. KreMLin is designed to emit well-formatted, idiomatic C code suitable for manual review. The resulting C programs can be compiled with CompCert for greatest assurance, and with mainstream C compilers, including GCC and Clang, for greatest performance. We have used KreMLin to extract to C the 20,000+ lines of Low⋆ code we have written so far. The performance of our verified code after KreMLin extraction is comparable to unverified, hand-written C code.

Our formal results cover the translation of standalone Low⋆ programs to C, proving that execution in C preserves the original F⋆ semantics of the Low⋆ program. More pragmatically, we have built several cryptographic libraries in Low⋆, compiled them to C, and integrated them within larger programs, allowing us to incrementally deploy our verified cryptographic libraries within other F⋆ developments that compile to OCaml (§4).

3 Proving Cryptographic Implementations in Low⋆

In this section, we sketch a few simple fragments of code from our Low⋆ implementation of the TLS-1.3 record layer. First, we illustrate functional correctness properties proven of an efficient implementation of the Poly1305 Message Authentication Code (MAC) algorithm [2]. Then, discuss our model of game-based cryptography in F⋆ and its use in proving security of the main authenticated encryption construction used in TLS-1.3.

3.1 Functional Correctness of Poly1305

Arithmetic for the Poly1305 MAC algorithm is performed modulo the prime $2^{130} - 5$, i.e., the algorithm works in the finite field $GF(2^{130} - 5)$. To specify modular arithmetic in this field in F⋆, we make use of refinement types, as shown below.

```
val p = 2^130 - 5 (* the prime order of the field *)
type elem = n:nat {n < p} (* abstract field element *)
let ( + ) (x y : elem) : elem = (x + y) % p (* field addition *)
let ( * ) (x y : elem) : elem = (x * y) % p (* field multiplication *)
```

This code uses F⋆ infinite-precision natural numbers (nat) to define the prime order p of the field and the type of field elements, elem, inhabited by natural numbers n smaller than p. It also defines two infix operators for addition and multiplication in the field in terms of arithmetic on nat. Their result is annotated with elem, to indicate that these operations return field elements. The F⋆ typechecker automatically checks that the result is in the field; it would report an error if, for example, we omitted the reduction modulo p. These operations are convenient to specify polynomial computations but are much too inefficient for deployable code.

Instead, typical 32-bit implementations of Poly1305 represent field elements as mutable arrays of 5 unsigned 32-bit integers, each holding 26 bits. This representation evenly spreads out the bits across the integers, so that carry-overs during arithmetic operations can be delayed. It also enables an efficient modulo operation for p. We show below an excerpt of the interface of our lower-level verified implementation, relying on the definitions above to specify their correctness.

```
1 abstract type repr = buffer UInt32.t 5 (* 5-limb representation *)
2 val '_.[_] ': memory → repr → Ghost elem (* m.[r] is the value of r in m *)
3 val multiply: e0:repr → e1:repr → ST unit (* e0 := e0 * e1 *)
4   (requires (λ m → e0 ∈ m ∧ e1 ∈ m ∧ disjoint e0 e1))
5   (ensures (λ m0 _ m1 → modifies {e0} m0 m1 ∧ m1.[e0] = m0.[e0] * m0.[e1]))
```

The type repr defines the representation of field elements as buffers (mutable arrays) of 5 32-bit integers. It is marked as abstract, to protect the representation invariant from the rest of the code.

Functions are declared with a series of argument types (separated by →) ending with a return type and an effect (e.g., Ghost, ST, etc.). Functions may have logical pre- and post-conditions that refer to their arguments, their result, and their effects on the memory. If they access buffers, they typically have a pre-condition requiring their caller to prove that the buffers are live in the current memory.

The term m.[r] selects the contents of a buffer r from a memory m; it is used in specifications only, as indicated by the Ghost effect annotation on the final arrow of its type on line 2. The multiply function is marked as ST, to indicate a stateful computation that may read, write and allocate state. In a computation type ST a (requires pre) (ensures post), a is the result type of the computation, pre is a predicate on the input state, an post is a relation between the input state, the result value, and the final state. ST computations are also guaranteed to not leak any memory. The specification of multiply requires that its arguments are live in the initial memory (m) and refer to non-overlapping regions of memory; it computes the product of its two arguments and overwrites e0 with the result. Its post-condition states that the value of e0 in the final memory is the field multiplication of its value in the initial memory with that of e1, and that multiply does not modify any other existing memory location.

Implementing and proving that multiply meets its mathematical specification involves hundreds of lines of source code, including a custom, verified Bignum library in Low* [16]. Using this library, we implement poly1305_mac and prove it functionally correct. Its specification below states that the final value of the 16 byte tag (h1.[tag]) is the value of Spec.mac_1305, a polynomial of the message and the key encoded as field elements.

```
1  val poly1305_mac:
2    tag:nbytes 16ul →
3    len:u32 → msg:nbytes len{disjoint tag msg} →
4    key:nbytes 32ul{disjoint msg key ∧ disjoint tag key} → ST unit
5    (requires (λ h → msg ∈ h ∧ key ∈ h ∧ tag ∈ h))
6    (ensures (λ h0 _ h1 →
7      let r = Spec.clamp h0.[sub key 0ul 16ul] in
8      let s = h0.[sub key 16ul 16ul] in
9        modifies {tag} h0 h1 ∧
10       h1.[tag] == Spec.mac_1305 (encode_bytes h0.[msg]) r s))
```

After verification, F* types and specifications are erased during compilation and the compiled code only performs efficient low-level operations. Indeed, after extraction by

KreMLin, our verified implementation of Poly1305 is just as fast as a widely used unverified implementation of the same algorithm in C [1], both consuming ≈ 2.2 cycles per byte.

3.2 Game-based Cryptography

Going beyond functional correctness, we sketch how we use Low* to do security proofs in the standard model of cryptography, using "authenticated encryption with associated data" (AEAD) as a sample construction. AEAD is the main protection mechanism for the TLS record layer; it secures most Internet traffic.

AEAD has a generic security proof by reduction to two core functionalities: a stream cipher (such as ChaCha20 [13]) and a one-time-MAC (such as Poly1305). The cryptographic, game-based argument supposes that these two algorithms meet their intended *ideal functionalities*, e.g., that the cipher is indistinguishable from a random function. Idealization is not perfect, but is supposed to hold against computationally limited adversaries, except with small probabilities, say $\varepsilon_{\text{ChaCha20}}$ and $\varepsilon_{\text{Poly1305}}$. The argument then shows that the AEAD construction also meets its own ideal functionality, except with probability say $\epsilon_{\text{Chacha20}} + \epsilon_{\text{Poly1305}}$.

To apply this security argument to our implementation of AEAD, we need to encode such assumptions. To this end, we supplement our real Low* code with ideal F* code. For example, ideal AEAD is programmed as follows:

- encryption generates a fresh random ciphertext, and it records it together with the encryption arguments in a log.
- decryption simply looks up an entry in the log that matches its arguments and returns the corresponding plaintext, or reports an error.

These functions capture both confidentiality (ciphertexts do not depend on plaintexts) and integrity (decryption only succeeds on ciphertexts output by encryption). Their behaviors are precisely captured by typing, using pre- and post-conditions about the ghost log shared between them, and abstract types to protect plaintexts and keys.

The abstract type of keys and the encryption function for idealizing AEAD is below:

```
type entry = cipher * data * nonce * plain
abstract type key = { key: keyBytes; log: if Flag.aead then ref (seq entry) else unit }
let encrypt (k:key) (n:nonce) (p:plain) (a:data)  =
  if Flag.aead
  then let c = random_bytes cipher_len in k.log := (c, a, n, p) :: k.log; c
  else encrypt k.key n p a
```

A module Flag declares a set of abstract booleans (*idealization flags*) that precisely capture each cryptographic assumption. For every real functionality that we wish to idealize, we branch on the corresponding flag.

This style of programming heavily relies on the normalization capabilities of F*. At verification time, flags are kept abstract, so that we verify both the real and ideal versions of the code. At extraction time, we reveal these booleans to be false. The F* normalizer then, e.g., drops the then branch, and replaces the log field with (), meaning that both the high-level, list-manipulating code and corresponding type definitions are erased, leaving only low-level code from the else branch to be extracted.

Using this technique, we verify by typing e.g. that our AEAD code, when using *any* ideal cipher and one-time MAC, perfectly implements ideal AEAD. We also rely on typing to verify that our code complies with the pre-conditions of the intermediate proof steps. Finally, we also prove that our code does not reuse nonces, a common cryptographic pitfall.

4 Ongoing work

4.1 Verified Assembly Language and Safe Interoperability with C

While Low⋆ and KreMLin provide reasonably efficient C-like implementations of cryptography, for the highest performance, cryptographic code often relies on complex hand-tuned assembly language that is customized for individual hardware platforms. Vale is a new DSL that supports foundational, automated verification of high-performance assembly code. The Vale tool transforms annotated assembly programs into deeply embedded term in a proof assistant, together with proofs that the term meets its specification. So far, we have used Dafny [9] as the embedding language for Vale and used this tool chain to verify the correctness and security of implementations of SHA-256 on x86 and ARM, and hardware-accelerated AES-CBC on x86. Several implementations meet or beat the performance of unverified, state-of-the-art cryptographic libraries.

In ongoing work, we have begun to use F⋆ as the embedding language for Vale, and are mechanizing a formal model of interoperability between Low⋆ and Vale. By defining the deeply embedded semantics of Vale in F⋆ as a Low⋆ interpreter for assembly language terms, we aim to show that invocations from C to assembly can be accounted for within a single semantics for both DSLs. A key challenge here is to reconcile Low⋆'s CompCert-like structured memory model with Vale's "big array of bytes" view of memory.

4.2 An Early Deployment of *Everest* within libcurl

Emphasizing the incremental deployment of our verified code, we have integrated our verified TLS record layer extracted from Low⋆ to C, as well as Vale implementations in assembly, within a new version of miTLS [3], implemented in F⋆, covering TLS-1.2 and TLS-1.3. Our eventual goal is for miTLS to be implemented in the Low⋆ subset of F⋆ and extracted to C, with functional correctness and security proofs. However, as of now, miTLS is only partially verified (the handshake being the main, remaining unverified component) and extracts to OCaml. However, already, by virtue of basic type safety, our partially verified version of miTLS provides safety against low-level attacks (e.g. HeartBleed) similar to other OCaml-based implementations of TLS [7].

Relying on OCaml's foreign function interface, we integrate OCaml extracted miTLS with the C-and-assembly extracted verified TLS record layer. Dually, we provide C bindings for calling into a miTLS layer which handles the socket abstraction, fragmenting buffers, etc. The result is a `libmitls.dll`, which we integrate within `libcurl`, a popular open-source library, used pervasively in many tools. This early integration allows us to use our verified software from within popular command line tools, like `git`, providing immediate benefits. At the moment, the throughput of `libmitls` is about 5× slower than OpenSSL. While the gap is significant, this level of performance represents an improvement of roughly two orders magnitude relative to prior, OCaml-only versions of miTLS. We hope to eliminate the remaining gap by migrating more of miTLS to Low⋆ and selected portions of performance-critical crypto algorithms to Vale.

5 Parting Thoughts

A significant novelty of our proposed work is that we aim to replace security-critical components of existing software with provably correct versions. Our careful choice of the problem domain is crucial: verified OS kernels and compilers can only succeed by replacing the software stack or development toolchain; verified, standardized, security protocols, and HTTPS and TLS in particular, can be deployed within the existing ecosystem, providing a large boost to software security at a small overall cost.

With the emergence of TLS 1.3, most TLS implementers, including Windows, will rewrite their implementations from scratch, and the world will be faced with migrating towards brand new implementations. History tells us that widespread adoption of a new version of TLS may take almost an entire decade. Given a similar time line for the adoption of TLS 1.3, if we distribute *Everest* within 2–4 years in a form where the cost of switching to it is negligible, then we are optimistic that it stands a chance of widespread adoption.

Despite this once-in-a-decade opportunity, several challenges remain. How will verified code authored in advanced programming languages be maintained and evolved going forward? Distributing our code as well-documented, source-like C may help somewhat, but to evolve the code while maintaining verification guarantees will require continued support from the *Everest* team, as well as outreach and education. Will the software industry at large be able to appreciate the technical benefits of verified code? How can we empirically "prove" that verified software is better? One direction may be to deploy, standalone, small-TCB versions of *Everest* and showing it to be resistant to attack—this raises the possibility of deployments of *Everest* within fully verified stacks [8, 6, 5] or sandboxes [14].

References

1. The sodium crypto library (libsodium). URL: https://www.gitbook.com/book/jedisct1/libsodium/details.
2. Daniel J Bernstein. The Poly1305-AES message-authentication code. In *International Workshop on Fast Software Encryption*, pages 32–49. Springer, 2005.
3. Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*, pages 445–459, 2013.
4. Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.
5. Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 653–669, Berkeley, CA, USA, 2016. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=3026877.3026928.
6. Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 165–181, Berkeley, CA, USA, 2014. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2685048.2685062.
7. David Kaloper-Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 223–238, 2015.

8 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1629575.1629596, doi:10.1145/1629575.1629596.

9 K. Rustan M. Leino. 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, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1939141.1939161.

10 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

11 Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, June 2012. URL: http://hal.inria.fr/hal-00703441.

12 Pierre Letouzey. A new extraction for Coq. In *Types for proofs and programs*, pages 200–219. Springer, 2002.

13 Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF protocols. IETF RFC 7539, 2015.

14 Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A design and verification methodology for secure isolated regions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 665–681, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2908080.2908113, doi:10.1145/2908080.2908113.

15 Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, 2016. URL: https://www.fstar-lang.org/papers/mumon/.

16 Jean Karim Zinzindohoué, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A verified extensible library of elliptic curves. In *IEEE Computer Security Foundations Symposium (CSF)*, 2016.