Practical Formal Methods for Real World Cryptography

Karthikeyan Bhargavan
Inria, Paris, France
karthikeyan.bhargavan@inria.fr

Prasad Naldurg
Inria, Paris, France
prasad.naldurg@inria.fr

Abstract

Cryptographic algorithms, protocols, and applications are difficult to implement correctly, and errors and vulnerabilities in their code can remain undiscovered for long periods before they are exploited. Even highly-regarded cryptographic libraries suffer from bugs like buffer overruns, incorrect numerical computations, and timing side-channels, which can lead to the exposure of sensitive data and long-term secrets. We describe a tool chain and framework based on the F∗ programming language to formally specify, verify and compile high-performance cryptographic software that is secure by design. This tool chain has been used to build a verified cryptographic library called HACL∗, and provably secure implementations of sophisticated secure communication protocols like Signal and TLS. We describe these case studies and conclude with ongoing work on using our framework to build verified implementations of privacy preserving machine learning software.

2012 ACM Subject Classification Security and privacy → Formal security models; Security and privacy → Logic and verification

Keywords and phrases Formal verification, Applied cryptography, Security protocols, Machine learning

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2019.1

Category Invited Talk

1 Introduction

Cryptography is the backbone of most internet applications, including e-commerce, online payment, messaging, social networking, and user communications. Different algorithms and protocols are used to guarantee different levels of confidentiality, integrity and authentication protection, depending on application and user requirements. In some applications, its use can be opaque to end users, such as in digital rights management and business analytics. While there is no need to motivate the use of cryptography online, implementing cryptographic software for real world applications can be incredibly complex and error-prone. Though governments, companies, and standards bodies have been using and stress-testing cryptographic algorithms for more than twenty years, surprisingly, there is a lack of rigour in how many new protocols and applications are implemented.

Implementations of cryptographic primitives can have obvious as well as subtle vulnerabilities that are often difficult to detect. To illustrate, in OpenSSL, a widely used open-source (and hence open to scrutiny) implementation of common cryptographic algorithms, 16 CVEs (common vulnerability and exposures reports) have been issued since 2017 for vulnerabilities in the core cryptographic functions. These bugs range from incorrect implementations of numerical computations (5), to timing side channel attacks (6), and memory safety issues (5). Such programming errors can often be exploited by a remote attacker to tamper with the cryptographic computation, leading to various degrees of exposure, and invalidating the

39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2019).
Editors: Arkadev Chattopadhyay and Paul Gastin; Article No. 1; pp. 1:1–1:12
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

security guarantees the algorithm was designed for in the first place. As a typical example, Brumley et al. [19] show how an arithmetic bug in the implementation of an elliptic curve in OpenSSL can be practically exploited to retrieve a victim's long term private key.

Finding such bugs in large codebases that are focused more on high-performance than high-assurance is not an easy task. Software development practices, from good hygiene and code reviews, to unit-testing and fuzzing, are best-effort and usually incapable of finding subtle vulnerabilities. Rather than attempt to find and fix bugs in an ad hoc manner, our philosophy, in line with a number of recent works [20, 7, 46, 15, 6, 39, 26], is to use formal verification to prove the absence of large classes of vulnerabilities by design.

We use the F∗ programming language and verification framework [43] to build HACL∗, a library of verified cryptographic algorithms in C. Given a published standard specification of a cryptographic primitive, we write verified code in F∗ that is memory safe, functionally correct, and resistant to timing side-channels. This code is then compiled to readable C code that is as performant as hand-written C code in state-of-the-art libraries like OpenSSL. HACL∗ supports most of the algorithms used in modern cryptographic protocols and applications, and is currently being used by the Mozilla Firefox Web browser, the WireGuard VPN, the Tezos Blockchain, and the Microsoft WinQuic protocol stack.

HACL∗ provides a robust basis for building high-assurance cryptographic applications, but the cryptographic library is only one component of the security stack. To protect connections between clients and servers, Web applications rely on standardized protocols like Transport Layer Security (TLS) [42]. For end-to-end secure messaging, WhatsApp and Skype rely on a complex cryptographic protocol called Signal [1]. These protocols invoke a series of cryptographic constructions across multiple messages to achieve sophisticated security goals. The overall security of each protocol depends on subtle invariants, which may be falsified by incorrect designs or buggy implementations. For example, the Triple Handshake attacks on the TLS [13] uncovered a protocol design flaw in the way three TLS sessions can be composed together, resulting in an attack on client authentication that had remained undiscovered for 18 years. The SMACK attacks on TLS libraries [9] found a class of implementation bugs that allowed attacker to completely bypass the security of a large subset of HTTPS connections on the Web. Preventing these kinds of attacks requires careful formal analysis.

We observe that cryptographic primitives are themselves getting more complex, with new post-quantum algorithms and homomorphic encryption constructions currently being standardized and deployed. Applications that use these new constructions, such as electronic voting and privacy preserving machine learning, are even more complicated to specify and analyse than traditional cryptographic protocols. Inevitably, attackers are also getting more sophisticated, and the classic network attacker model needs to be augmented with finer distinctions to catch and fix vulnerabilities.

We argue that the combination of complex protocols, sophisticated security properties, and powerful attackers demands a more rigorous treatment of cryptographic software development. In this paper, we describe how we can apply our verification tool chain across all layers of a secure distributed application, starting with cryptographic algorithms (Section 2), to end-to-end protocols with sophisticated security properties (Section 3), all the way to novel privacy-preserving applications (Section 4). Through these case studies, we show how formal methods can play an important role in building high-assurance cryptographic software.

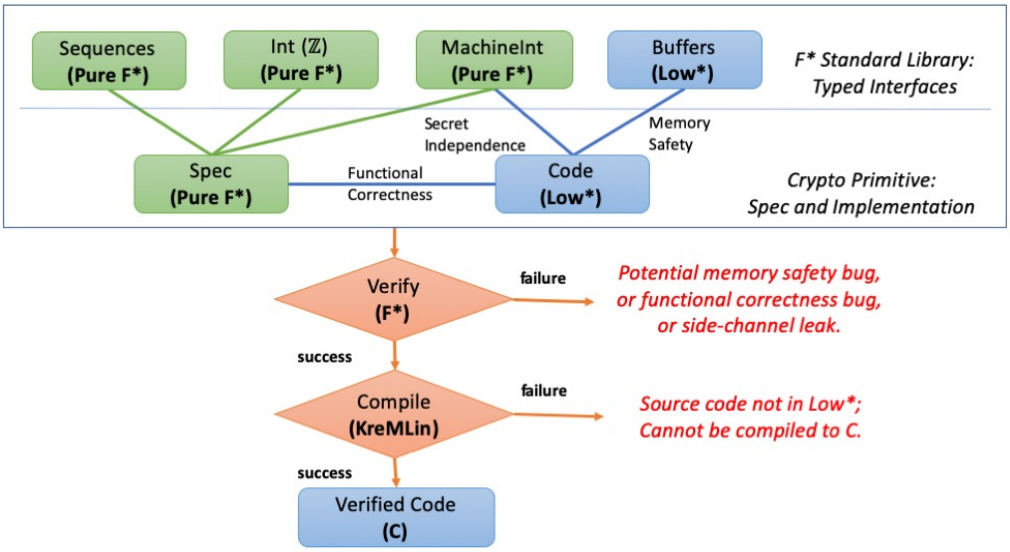

Figure 1 HACL∗ verification and compilation tool chain.

2 Verified Cryptography: HACL∗

HACL∗ [47] is a verified open-source library of modern cryptographic algorithms, including the elliptic curve Curve25519 [3], the authenticated encryption construction ChaCha20-Poly1305 [2], the hash function SHA-2 [44], and the signature scheme Ed25519 [4]. Put together, these algorithms are enough to satisfy all the classic cryptographic needs of a distributed software application. In particular, HACL∗ supports the full NaCl cryptographic API [8], and implements a full ciphersuite of TLS 1.3 [42]. The distributable code of HACL∗ is in portable C, which can be easily wrapped into multiple languages and dropped into application software that needs these algorithms. For example, HACL∗ is currently used to implement TLS in Mozilla Firefox and as the NaCl implementation in the Tezos blockchain.

The verification and compilation tool chain used in the development of HACL∗ is depicted in Figure 1. All the code in HACL∗ is written in F∗, an ML-like functional programming language with a type system that includes polymorphism, dependent types, monadic effects, refinement types, and a weakest precondition calculus [43]. The language is aimed at program verification, and its type system allows the expression of precise and compact functional correctness and security property specifications for programs, which can be mechanically verified, with the help of an SMT solver. After verification, an F∗ program can be compiled to OCaml, F#, C, or even WebAssembly, and so it can run on a variety of platforms.

Figure 1 shows the workflow for adding a new verified cryptographic primitive in HACL∗. The first step is to write a high-level specification (Spec) in a higher-order purely functional subset of F∗. This specification relies on standard libraries for basic datatypes such as mathematical and machine integers ($\mathbb{Z}$,MachineInt), and immutable arrays (Sequences), also written in Pure F∗. Next, an optimized implementation of the primitive itself (Code) is written in Low∗, a low level subset of F∗ that can be efficiently compiled to C, using the KreMLin compiler [41]. For a full description of the syntax, type system, and semantics of F∗, refer to [43], and for the formal development of Low∗ and its compilation to C, see [41].

The Low∗ Code cannot use mathematical integers, and it is only allowed to use machine integer operations in ways that are safe from timing side channels. For example, if an unsigned 32-bit integer (uint32) holds a secret value, e.g. part of an encryption key, it cannot be compared with another integer, it cannot be used as an index into an array, and it cannot be used in a division or modulo operation. This is because, on most hardware platforms, the time taken by these operations may reveal the contents of the secret integer to a remote attacker. Cryptographic code that uses such operations is not *secret independent*, and hence may be vulnerable to various side-channels attacks.

The Low∗ code can also use mutable but memory-safe arrays (Buffers) to hold cryptographic state. However, all the arrays used in HACL∗ are stack-allocated, that is, they never use the heap, and hence do not have to be explicitly allocated or freed.

The code for each crytpographic algorithm is then verified, using the F∗ typechecker, to ensure that it conforms with the logical preconditions and type abstractions in the F∗ library. A failure to type-check here may indicate the presence of memory safety, functional correctness, or side channel vulnerabilities (or that the type checker may need more annotations to prove correctness). If type checking succeeds, the Low∗ code is compiled using KreMLin to portable C code, preserving all the properties verified in F∗.

Surprisingly, writing formally verified cryptographic code in HACL∗ does not have a performance cost. Our C code is as fast as the hand-optimized C code in state-of-the-art cryptographic libraries like OpenSSL. In many cases, the structured compact code generated from F∗ is even faster. Performance is especially important for encryption algorithms and elliptic curves that are used within network protocols like TLS, where cryptography often dominates cost and can be a performance bottleneck. For example, our HACL∗ implementation of Curve25519 was about 20% faster than the previous code for this elliptic curve in Firefox. Hence adopting our code significantly cut the cost of HTTPS connections between Firefox and popular websites like GMail. Similarly, the WireGuard VPN [24], which runs within the Linux Kernel and needs high-performance high-assurance code for Curve25519, uses HACL∗.

HACL∗ is an evolving project. We are extending it with more elliptic curves, encryption algorithms, and hash functions, and use it as a basis for building implementations of more advanced and experimental cryptographic constructions including post-quantum cryptography and homomorphic encryption. As a part of our privacy preserving machine learning project, which we describe in Section 4, we are building verified implementations of several partially homomorphic encryption schemes including Goldwassser-Micali [29], the Paillier [36] additive homomorphic system, and the DGK system [22, 23], using the BigNum library and other verified primitives from HACL∗. To further improve the performance of HACL∗ code, we are building a cryptographic provider called EverCrypt that combines verified C code from HACL∗ with verified assembly code from the Vale project [15].

3 Verified Protocols: LibSignal∗

In this section, we describe how to extend the scope of our security guarantees from cryptographic libraries to cryptographic protocols. Protocols that are built using verified primitives are not automatically secure, and require a different set of tools for specification and verification of higher-level properties.

We illustrate this with our work on Signal, an end-to-end encryption protocol for instant messaging that is used in many popular applications like WhatsApp, Skype, and Facebook Messenger, by billions of users worldwide. The main design goal of Signal is to maximally protect the privacy of its users, even if the Signal servers are compromised, and even if some

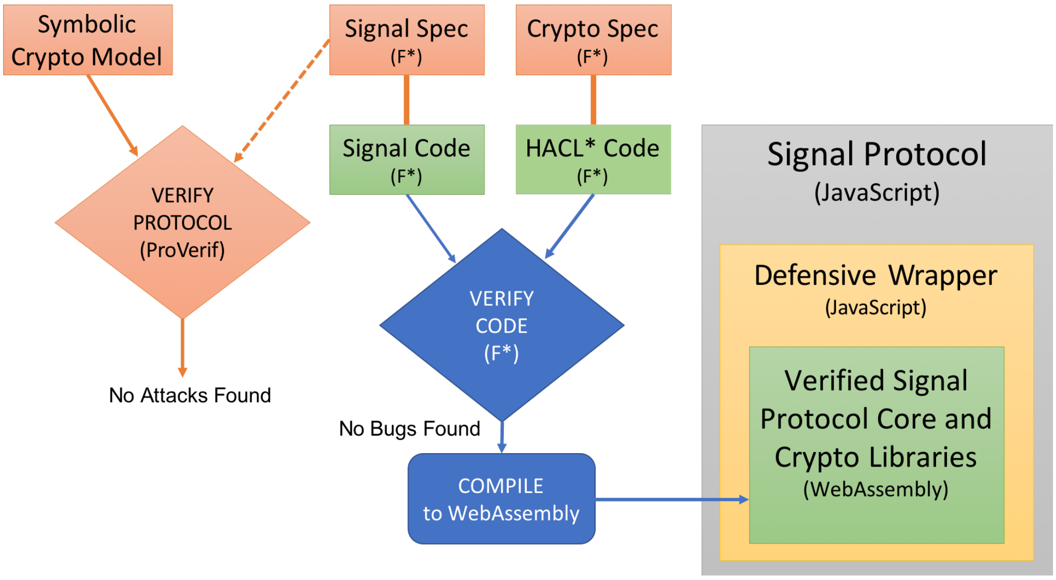

Figure 2 LibSignal* verification and compilation toolchain.

user devices are stolen or confiscated. To this end, Signal uses a novel key exchange protocol called X3DH [35] paired with an aggressive key update mechanism called Double Ratchet [38] that frequently changes message encryption keys, rendering old keys obsolete. Formally, Signal seeks to achieve a novel property called *post-compromise security* [21], in addition to classic secure channel guarantees like sender authentication, message confidentiality, and forward secrecy.

There are several implementations of Signal, including official libraries in Java (for Android phones), in C (for iPhones), and in JavaScript (for Web applications), that are embedded within various messaging applications. For example, the desktop version of Skype uses a library called libsignal-javascript for private conversations. This means that any flaw in the design of Signal or a bug in its JavaScript code may break the security of these private conversations.

We have built a verified implementation of Signal called LibSignal∗ [40] using the tool chain depicted in Figure 2. Note the similarity in the overall work flow with our tool chain in Figure 1. We first write a formal specification of the Signal protocol in the pure fragment of F∗. We then hand-translate this specification to the syntax of the ProVerif protocol analyzer [14] and verify it for all the target security properties of Signal, including forward and post-compromise security, following the methodology of [33]. If ProVerif fails to verify the protocol, it produces a counter-example that may indicate a security vulnerability. However, our analysis found no flaws in Signal, except for a known replay vulnerability [33].

Our next step was to write a Low∗ implementation of Signal, which needed several cryptographic algorithms, including AES-CBC, HMAC, Curve25519, Ed25519, and SHA-2, all of which we implement and verify in HACL∗. We then verify the Low∗ code of Signal (composed with the Low∗ code for HACL∗) for conformance to the high-level protocol specification. Finally, we compile the code, via the KreMLin compiler to C and WebAssembly, obtaining verified implementations of the Signal protocol in these languages.

WebAssembly [31] is a new meta-assembly language supported by all Web browsers and many application frameworks. It allows compact and efficient low-level programs to be embedded within JavaScript applications and run on any platform. In comparison to

JavaScript, WebAssembly enjoys many advantages, making it a good target for verified code. In particular, WebAssembly is a small, statically typed language with a clean formal semantics, and it offers strong isolation guarantees against malicious JavaScript code. We develop a formal translation from Low∗ to WebAssembly and implement this as a new back-end for the KreMLin compiler [40]. We use this back-end to compile both HACL∗ and LibSignal∗. Our WebAssembly version of HACL∗ may independently be used in any JavaScript application that needs verified cryptography.

We observe that just generating the core cryptographic protocol code for Signal does not make it immediately usable by a messaging application. For example, the libsignal-protocol-javascript library provides a session and key management layer and exposes a simple interface to its applications. Our implementation of LibSignal∗ borrows this JavaScript code so that we meet the same interface and pass all the interoperability tests of Signal. Notably, however, we embed our verified WebAssembly code into the unverified JavaScript in a defensive manner that reduces the risk of private key exposure.

Our work in Signal is also influenced by our experience with the verification of the Transport Layer Security (TLS) protocol, the de facto standard for secure communications across the Internet. Although TLS was carefully specified and widely implemented, a large number of vulnerabilities were regularly found, both in the protocol design (e.g [13]) and in its implementations (e.g. [9]). When the Internet Engineering Task Force (IETF) began the process of standardizing TLS 1.3, it invited researchers to help them design the new protocol to be secure by design. Many researchers responded to this challenge, publishing a series of papers analyzing various draft versions of the protocol. In our work, we built detailed formal models of several drafts of TLS 1.3 using the verification tools ProVerif and CryptoVerif [10]. As part of Project Everest [11], we are also helping build a verified implementation of TLS 1.3 in F∗ using the same tool chain as HACL∗, but extending it with cryptographic security proofs [12].

Our work with LibSignal∗ and TLS shows how we can compose the low-level guarantees of HACL∗ with the sophisticated security proofs of ProVerif and other tools to obtain a verified cryptographic protocol implementation that can readily be deployed in real world messaging applications. We believe that this methodology offers a template for many more future applications.

4 Verified Applications

Encouraged by this flexibility and modularity, we plan to extend our framework to target distributed web applications beyond cryptographic primitives and communication protocols. We describe this next in the context of preserving privacy in machine learning classification, where its secure implementation will require certification of application code on clients and servers.

4.1 Privacy Preserving Machine Learning

Machine learning classification as-a-service is an attractive use-case for cloud servers. Such a server would host a classifier algorithm, and process and reply to classification queries from authenticated subscribers (or clients). Since learning applications consume large amounts of training data to generate useful classifiers, user privacy is a pressing concern. Protecting sensitive and personally identifiable information (PII) of users from servers, both during model learning and subsequent classification is desirable and can be a legal/compliance requirement. We focus only on the classification phase here to illustrate our techniques.

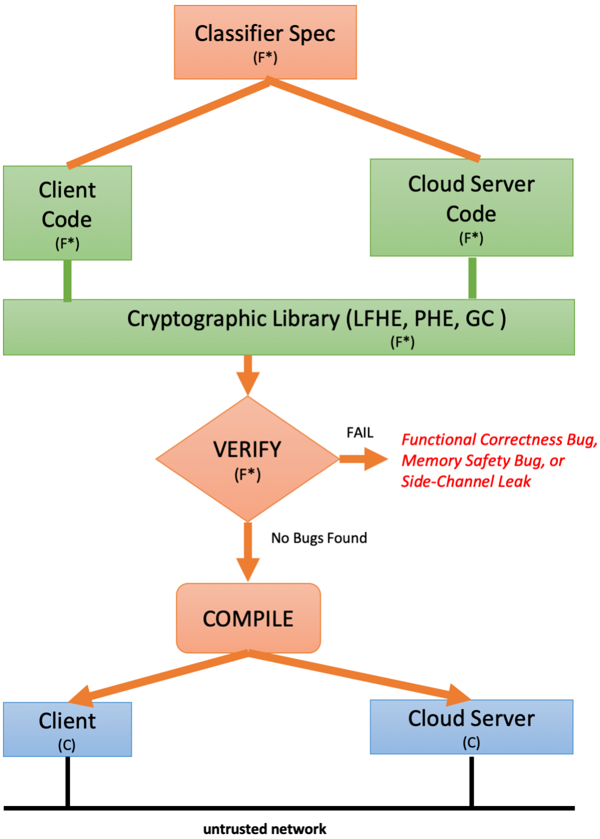

Figure 3 Programming and Verification Framework: The programmer first writes a high-level mathematical specification of the classifier (or any other computation over private data) in F∗. The programmer can run and test this specification. She then implements this specification as a distributed program with components running at the client and the cloud server. The program is composed with a cryptographic library and the whole system is verified using F∗. If verification succeeds, the code is compiled to C and can be deployed on the network.

A machine-learning classifier that preserves user privacy should not learn anything about the user query issued by a client or its resulting response (i.e.,the resulting class). At the same time, from the point of view of server, the mathematical models used for learning and inference can be proprietary and need to be hidden from clients.

We describe the context of our work in more detail. In model learning, the inputs to the learning algorithm are labeled data values, converted to feature vectors $\vec{x}$, and used to learn a model of weights w of a classifier consisting of say k classes $c_1 \cdots c_k$, given by $C(\vec{x}, w)$. In the *classification* phase, the label $c_j, 1 \leq j \leq k$ for an unseen feature vector $\vec{y}$ input by a client, is predicted using the classifier C as $c_j = C(\vec{y}, w)$. As mentioned earlier, we focus only on the classification phase, where the server is presented with a query and is expected to return the appropriate class label prediction.

Cryptographic techniques can offer a solution to this problem that satisfies both parties. Some relevant cryptographic schemes in this context include applications based on homomorphic encryption (HE) [27, 18, 32, 30] secure multi-party computation (SMC) [45, 34], garbled circuits [30, 32], and functional encryption (FE) [17], which allow clients and servers to jointly compute functions over encrypted or private data without revealing their inputs to each other. In HE, the result also remains encrypted, and can only be decrypted with the appropriate key. A typical HE algorithm takes an encrypted input x for program P and produces the encrypted result of applying x on the function encoded by P.

With HE, both the model w and query $\vec{y}$ are encrypted using say a public HE key. The prediction classifier is implemented on the server as the homomorphic evaluation function $Eval(C)$. The result of the prediction, c_j, has to be *declassified* and presented to the client that issued the query. The cryptographic properties of the HE scheme ensure that the client does not learn anything about model w beyond what it can learn from observing the predicted class of its input, and the server does not learn the value of the input, or its predicted class. A caveat here is that there are certain types of attacks, including model inversion, and access to prior knowledge, that can reveal user information even if they are encrypted. Techniques such as differential privacy [25] can help alleviate these concerns, and we plan to study them in the future.

HE schemes that can compute arbitrary functions (called fully HE or FHE) are fairly straightforward to implement, but are prohibitively expensive. Even with the latest implementation of HELib [28], general depth-limited homomorphic computations of interest in machine learning have very large overheads, e.g, with matrix multiplication being over 600K times slower than plaintext computations, which does not make them practical for useful applications. However, HE schemes that are restricted in their functionality, called partial HE schemes (PHE) are more practical, and can perform one type, say add or multiply [37, 29] or a small number of computations, e.g., quadratic functions [16]. We have seen e.g., in [18, 32], that PHE schemes can be combined with other auxiliary cryptographic schemes such as secure multi-party computations (SMC) and garbled circuits (GC), or even with strong hardware protection guarantees to build solutions that are practical, and provide strong guarantees.

We propose a programming and verification framework to help developers build distributed software applications using composite PHE protocols, and extend it to include auxiiary schemes such as SMC and GCs, incorporating verified cryptographic primitives and their high-performance implementations. With our framework, a developer can prove that the application code is functionally correct, that it correctly composes the various cryptographic schemes and protocols it uses, and that it does not accidentally leak any secrets (via side-channels, for example.) Our end-to-end solution can be seen as a logical extension of our work presented in the earlier two sections, and results in verified and efficient implementations of state-of-the-art secure privacy-preserving learning and classification.

Given a high-level algorithmic specification of a machine learning classifier, along with a set of confidentiality constraints on its inputs, our goal is to build and verify its implementation as an efficient distributed cryptographic application. Our verified implementation tool chain is shown in Figure 3, with four stages. Again, note how this can be seen as a modular extension of our earlier designs.

1. **Global High-Level Specification**: We first write a global high-level specification of our desired distributed computation in F∗, focusing on classification algorithms for now. The specification consists of the function ϕ it computes, the characterization of its model w, in terms of feature vectors $\vec{\chi}$, input $\vec{x} \in \vec{\chi}$, and the result $c_i = \phi(w, \vec{x})$ from $\mathcal{C}$ the set of classes. The high-level confidentiality specification is that the evaluation of ϕ must preserve the secrecy of w, $\vec{x}$, and c_i from relevant parties.
2. **Distributed Implementation**: We then write implementations, also in F∗, of the client and the cloud server, detailing all their network interactions and cryptographic computations. We prove that this implementation meets the high-level spec, while

preserving our desired confidentiality goals, given an abstract (trusted) interface for the underlying cryptography. The implementation can itself be broken into a reusable verified library of commonly used constructions, like addition, secure comparison, dot products, polynomial evaluation, etc. and application-specific code for the classification algorithm we seek to implement.

3. **Cryptographic Instantiation**: The code for these two parties will usually rely on a variety of cryptographic primitives, which will need to be instantiated with concrete schemes such as Paillier, GCs, random permutations, etc. which are themselves hard to implement correctly. We build verified implementations of all the cryptographic schemes we need, as an extension to the HACL∗ verified crypto library. These primitives compile to C code that is as fast as state-of-the-art hand-written crypto libraries. Each primitive is verified for memory safety, resistance to common timing side-channels, and functional correctness with respect to a high-level mathematical specification. We propose to build a series of verified HE and SMC schemes in HACL∗, which will also be reusable in other applications.
4. **Low-Level Executable Components**: Finally, we compile all our F∗ code along with the cryptographic library to C to obtain two C implementations, one for the client and one for the server. We envisage that these libraries will be embedded into larger applications that will handle less security-critical concerns like user interfaces, networking code, and persistent storage. Generating C code allows our code to run efficiently on a variety of platforms, including smartphones, and enables existing legacy applications to use our toolchain to verify their core cryptographic components.

At the end of this workflow, we obtain high performance verified protocol code in C for clients, and servers which can communicate over an untrusted network, but still provide strong correctness and confidentiality guarantees.

5 Proposed Roadmap

We propose to build our verification toolchain in stages, evaluating them over a series of case studies. Our eventual goal is to be able to verify privacy-preserving implementations of inference for naïve Bayes classifiers, hyperplane decision classifiers (perceptron, least squares, Fischer's linear discriminants, SVMs), decision tree classifiers, and neural networks.

As a longer-term goal, we see our toolchain as something that can be integrated into a mainstream framework for building distributed cryptographic applications. For example, the machine learning framework can be integrated with TensorFlow [5], which offers an API to developers that is not very far from the core operations we consider in this work: addition, multiplication, dot-product, comparison etc. We envision that machine learning developers will be able to write and test their high-level specifications as TensorFlow programs and our toolchain will help them develop verified low-level distributed protocols that implement these programs in a privacy-preserving style that can be safely deployed in the untrusted cloud.

Our verified framework and the modular tool chain allows us to develop high-assurance cryptographic applications that incorporate state-of-the-art cryptographic algorithms, complicated cryptographic protocols and their composition, and allow us analyze the resulting implementation for sophisticated and fine-grained end-to-end security properties.

References

1 Signal. `https://signal.org/docs/`.

2 ChaCha20 and Poly1305 for IETF Protocols. IETF RFC 7539, 2015.

3 Elliptic Curves for Security. IETF RFC 7748, 2016.

4 Edwards-Curve Digital Signature Algorithm (EdDSA) . IETF RFC 8032, 2017.

5 Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: `http://tensorflow.org/`.

6 José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-Assurance and High-Speed Cryptography. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1807–1823, 2017.

7 Andrew W Appel. Verification of a cryptographic primitive: SHA-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(2):7, 2015.

8 Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 159–176. Springer, 2012.

9 Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A Messy State of the Union: Taming the Composite State Machines of TLS. In *IEEE Symposium on Security and Privacy*, pages 535–552, 2015.

10 Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *IEEE Symposium on Security and Privacy*, pages 483–503, 2017.

11 Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pang, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *Summit on Advances in Programming Languages (SNAPL)*, 2017.

12 Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jianyang Pan, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, and Jean Karim Zinzindohoué. Implementing and Proving the TLS 1.3 Record Layer. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2017.

13 Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS. In *IEEE Symposium on Security and Privacy*, pages 98–113, 2014.

14 Bruno Blanchet. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1-2):1–135, October 2016.

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

16 Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF Formulas on Ciphertexts. In *Theory of Cryptography Conference (TCC)*, pages 325–341, 2005.

17 Dan Boneh, Amit Sahai, and Brent Waters. Functional Encryption: Definitions and Challenges. In *Theory of Cryptography Conference (TCC)*, pages 253–273, 2011.

18 Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine Learning Classification over Encrypted Data. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

19 Billy B Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ECC-related software bug attack. In *Topics in Cryptology (CT-RSA)*, pages 171–186. Springer, 2012.

20 Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. Verifying Curve25519 Software. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 299–309, 2014.

21 Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On Post-compromise Security. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 164–178, 2016.

22 Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Efficient and Secure Comparison for On-Line Auctions. In Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson, editors, *Information Security and Privacy*, 2007.

23 Ivan Damgard, Martin Geisler, and Mikkel Kroigard. A Correction to “Efficient and Secure Comparison for On-Line Auctions”. *Int. J. Appl. Cryptol.*, 1(4), August 2008.

24 Jason A. Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

25 Cynthia Dwork. Differential Privacy. In *International Conference on Automata, Languages and Programming (ICALP) - Volume Part II*, 2006.

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

27 Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, Stanford, USA, 2009.

28 Craig Gentry and Shai Halevi. Implementing Gentry’s Fully-homomorphic Encryption Scheme. In *Advances in Cryptology (EUROCRYPT)*, pages 129–148, 2011.

29 Shafi Goldwasser and Silvio Micali. Probabilistic Encryption & How to Play Mental Poker Keeping Secret All Partial Information. In *ACM Symposium on Theory of Computing (STOC)*, 1982.

30 Trinabh Gupta, Henrique Fingler, Lorenzo Alvisi, and Michael Walfish. Pretzel: Email Encryption and Provider-supplied Functions Are Compatible. In *Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 169–182, 2017.

31 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web Up to Speed with WebAssembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 185–200, 2017.

32 Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security Symposium*, pages 1651–1669, 2018.

33 Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *IEEE European Symposium on Security and Privacy (EuroSP)*, pages 435–450, 2017.

34 Eleftheria Makri, Dragos Rotaru, Nigel P. Smart, and Frederik Vercauteren. EPIC: Efficient Private Image Classification (or: Learning from the Masters). Topics in Cryptology (CT-RSA), 2019.

35 Moxie Marlinspike and Trevor Perrin. The X3DH Key Agreement Protocol, 2016. URL: `https://signal.org/docs/specifications/x3dh/`.

36 Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238, 1999.

37 Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology (EUROCRYPT)*, pages 223–238, 1999.

38 Trevor Perrin and Moxie Marlinspike. The Double Ratchet Algorithm, 2016. URL: `https://signal.org/docs/specifications/doubleratchet/`.

39 Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying Arithmetic Assembly Programs in Cryptographic Primitives. In *Conference on Concurrency Theory (CONCUR)*, 2018.

40 Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally Verified Cryptographic Web Applications in WebAssembly. In *IEEE Symposium on Security and Privacy*, pages 1256–1274, 2019.

41 Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified Low-Level Programming Embedded in F*. *PACMPL*, 1(ICFP):17:1–17:29, September 2017.

42 E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3, 208. IETF RFC 8446.

43 Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270, 2016.

44 National Institute of Standards US Department of Commerce and Technology (NIST). Federal Information Processing Standards Publication 180-4: Secure hash standard (SHS), 2012.

45 Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: Efficient and Private Neural Network Training. In *Privacy Enhancing Technologies Symposium.* (PETS 2019), 2019.

46 Jean Karim Zinzindohoue, Evmorfia-Iro Bartzia, and Karthikeyan Bhargavan. A Verified Extensible Library of Elliptic Curves. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 296–309, 2016.

47 Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1789–1806, 2017.