Continuous formal verification of Amazon s2n

Andrey Chudnov[1], Nathan Collins[1], Byron Cook[3,4], Joey Dodds[1], Brian Huffman[1], Colm MacCárthaigh[3], Stephen Magill[1], Eric Mertens[1], Eric Mullen[2], Serdar Tasiran[3], Aaron Tomb[1], and Eddy Westbrook[1]

[1] Galois, Inc.
[2] University of Washington
[3] Amazon Web Services
[4] University College London

Abstract. We describe formal verification of s2n, the open source TLS implementation used in numerous Amazon services. A key aspect of this proof infrastructure is continuous checking, to ensure that properties remain proved during the lifetime of the software. At each change to the code, proofs are automatically re-established with little to no interaction from the developers. We describe the proof itself and the technical decisions that enabled integration into development.

1 Introduction

The Transport Layer Security (TLS) protocol is responsible for much of the privacy and authentication we enjoy on the Internet today. It secures our phone calls, our web browsing, and connections between resources in the cloud made on our behalf. In this paper we describe an effort to prove the correctness of s2n [?], the open source TLS implementation used by many Amazon and Amazon Web Services (AWS) products (*e.g.* Amazon S3 [?]). Formal verification plays an important role for s2n. First, many security-focused customers (*e.g.* financial services, government, pharmaceutical) are moving workloads from their own data centers to AWS. Formal verification provides customers from these industries with concrete information about *how* security is established in Amazon Web Services. Secondly, automatic and continuous formal verification facilitates rapid and cost-efficient development by a distributed team of developers.

In order to realize the second goal, verification must continue to work with low effort as developers change the code. While fundamental advances have been made in recent years in the tractability of full verification, these techniques generally either: 1) target a fixed version of the software, requiring significant re-proof effort whenever the software changes or, 2) are designed around synthesis of correct code from specifications. Neither of these approaches would work for Amazon as s2n is under continuous development, and new versions of the code would not automatically inherit correctness from proofs of previous versions.

To address the challenge of program proving in such a development environment, we built a proof and associated infrastructure for s2n's implementations of DRBG, HMAC, and the TLS handshake. The proof targets an existing implementation and is updated either automatically or with low effort as the code changes. Furthermore, the proof connects with existing proofs of security properties, providing a high level of assurance.

Our proof is now deployed in the continuous integration environment for s2n, and provides a distributed team of developers with repeated proofs of the correctness of s2n even as they continue to modify the code. In this paper, we describe how we structured the proof and its supporting infrastructure so that the lessons we learned will be useful to others who address similar challenges.

Figure **??** gives an overview of our proof for s2n's implementation of the HMAC algorithm and the tooling involved. At the left is the ultimate security property of interest, which for HMAC is that if the key is not known, then HMAC is indistinguishable from a random function (given some assumptions on the underlying hash functions). This is a fixed security property for HMAC and almost never changes (a change would correspond to some new way of thinking about security in the cryptographic research community). The HMAC specification is also fairly static, having been updated only once since its publication in 2002[5]. Beringer et al. [**?**] have published a mechanized formal proof that the high-level HMAC specification establishes the cryptographic security property of interest.

As we move to the right through Figure **??**, we find increasingly low-level artifacts and the rate of change of these artifacts increases. The low-level HMAC specification includes details of the API exposed by the implementation, and the implementation itself includes details such as memory management and performance optimizations. This paper focuses on verifying these components in a manner that uses proof automation to decrease the manual effort required for ongoing maintenance of these verification artifacts. At the same time, we ensure that the automated proof occurring on the right-hand side of the figure is linked to the stable, foundational security results present at the left.

In this way, we realize the assurance benefit of the foundational security work of Beringer et al. while producing a proof that can be integrated into the development workflow. The proof is applied as part of the *continuous integration* system for s2n (which uses Travis CI) and runs every time a code change is pushed or a pull request is issued. In one year of code changes only three manual updates to the proof were required.

The s2n source code, proof scripts, and access to the underlying proof tools can all be found in the s2n GitHub [**?**] repository. The collection of proof runs is logged and appears on the s2n Travis CI page [**?**].

In addition to the HMAC proof, we also reused the approach shown in the right-hand side of Figure **??** to verify the deterministic random bit generator (DRBG) algorithm and the TLS Handshake protocol. In these cases we didn't link to foundational cryptographic security proofs, but nonetheless had specifications that provided important benefits to developers by allowing them to 1) check their code against an independent specification and 2) check that their

[5] And this update did not change the functional behavior specified in the standard.

code continues to adhere to this specification as it changes. Our TLS Handshake proof revealed a bug (which was promptly fixed) in the s2n implementation [?], providing evidence for the first point. All of our proofs have continued to be used in development since their introduction, supporting the second point.

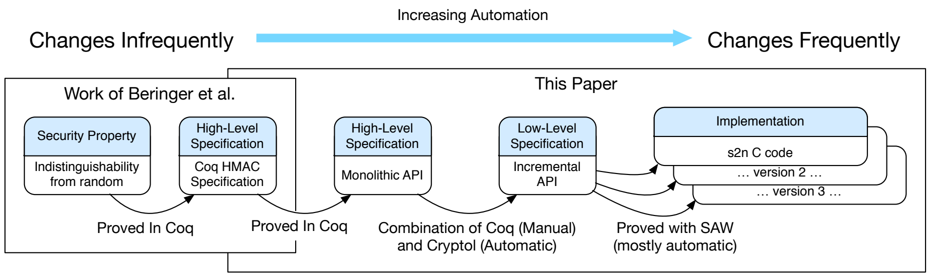

Fig. 1. An overview of the structure of our HMAC proof.

Related work. Projects such as Everest [?,?], Cao [?], and Jasmin [?], generate verified cryptographic implementations from higher level specifications, *e.g.* F* models. While progress in this space continues to be promising—HACL* has recently achieved performance on primitives that surpasses handwritten C [?]—we have found in our experiments that the generated TLS code does not yet meet the performance, power, and space constraints required by the broad range of AWS products that use s2n.

Static analysis for hand-written cryptographic implementations has been previously reported in the context of Frama-C/PolarSSL [?], focusing on scaling memory safety verification to a large body of code. Additionally, unsound but effective bug hunting techniques such as fuzzing have been applied to TLS implementations in the past [?,?]. The work we report on goes further by proving behavioral correctness properties of the implementation that are beyond the capabilities of these techniques. In this we were helped because the implementation of s2n is small (less than 10k LOC), and most iteration is bounded.

The goal of our work is to verify deep properties of an existing and actively developed open source TLS implementation that has been developed for both high performance and low power as required by a diverse range of AWS products. Our approach was guided by lessons learned in several previous attempts to prove the correctness of s2n that either (1) required too much developer interaction during the modification of the code [?], or (2) where pushbutton symbolic model checking tools did not scale. Similarly, proofs developed using tools from the Verified Software Toolchain (VST) [?] are valuable for establishing the correctness and security of specifications, but are not sufficiently resilient to code changes, making them challenging to integrate into an ongoing development process. Their use of a layered proof structure, however, provided us with a specification that we could use to leverage their security proof in our work.