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 [3], the open source TLS implementation used by many Amazon and Amazon Web Services (AWS) products (*e.g.* Amazon S3 [2]). 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 1 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. [6] 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 1, 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 [3] repository. The collection of proof runs is logged and appears on the s2n Travis CI page [4].

In addition to the HMAC proof, we also reused the approach shown in the right-hand side of Figure 1 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 code continues to adhere to this specification as it changes. Our TLS Handshake

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

proof revealed a bug (which was promptly fixed) in the s2n implementation [10], providing evidence for the first point. All of our proofs have continued to be used in development since their introduction, supporting the second point.

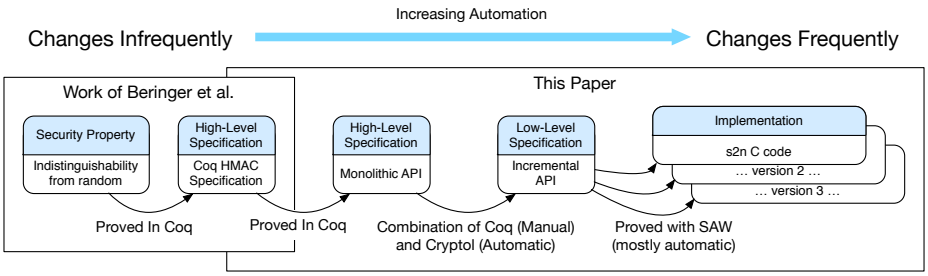

Fig. 1. An overview of the structure of our HMAC proof.

Related work. Projects such as Everest [8,12], Cao [5], and Jasmin [1], 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 [25]—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 [23], 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 [11,18]. 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 [17], or (2) where pushbutton symbolic model checking tools did not scale. Similarly, proofs developed using tools from the Verified Software Toolchain (VST) [6] 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.

O'Hearn details the industry impact of continuous reasoning about code in [19], and describes additional instances of integration of formal methods with developer workflows.

2 Proof of HMAC

In this section, we walk through our HMAC proof in detail, highlighting how the proof is decomposed, the guarantees provided, the tools used, and how this approach supports integration of verification into the development work-flow. While HMAC serves as an example, we have also performed a similar proof of the DRBG and TLS Handshake implementations. We do not discuss DRBG further, as there are no proof details that differ significantly from HMAC. We describe our TLS verification in Section 3.

2.1 High-level HMAC Specification

The keyed-Hash Message Authentication Code algorithm (HMAC) is used for authenticated integrity in TLS 1.2. Authenticated integrity guarantees that the data originated from the sender and was not changed or duplicated in transit. HMAC is used as the foundation of the TLS Pseudorandom Function (PRF), from which the data transmission and data authentication shared keys are derived. This ensures that both the sender and recipient have exchanged the correct secrets before a TLS connection can proceed to the data transmission phase.

HMAC is also used by some TLS cipher suites to authenticate the integrity of TLS records in the data transmission phase. This ensures, for example, that a third party watching the TLS connection between a user and a webmail client is unable to change or repeat the contents of an email body during transmission. It is also used by the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) which is implemented within s2n as a utility function for general purpose key derivation and is central to the design of the TLS1.3 PRF.

FIPS 198-1 [24] defines the HMAC algorithm as

$$\mathsf{HMAC}(K, message) = \mathsf{H}((K \oplus opad) \| \mathsf{H}((K \oplus ipad) \| message))$$

where H is any hash function, $\oplus$ is bitwise xor, and $\|$ is concatenation. *opad* and *ipad* are constants defined by the specification. We will refer to this definition as the *monolithic* specification.

Following Figure 1, we use the Cryptol specification language [14] to express HMAC in a form suitable for mechanized verification, first in a monolithic form, and then in an incremental form. We prove high-level properties with Coq [22] and tie these to the code using the Software Analysis Workbench (SAW) [16]. We first describe the proof of high-level properties before going into specifics regarding the tools in Section 2.4.

2.2 Security Properties of HMAC

The Cryptol version of the Monolithic HMAC specification follows.

```
hmac k message = H((k ^ opad) # H((k ^ ipad) # message))
```

where `H` is any hash function, `^` is bitwise xor, and `#` is concatenation.

The high-level Cryptol specification and the FIPS document look nearly identical, but what assurance do we have that either description of the algorithm is cryptographically secure? We can provide this assurance by showing that the Cryptol specification establishes one of the security properties that HMAC is intended to provide—namely, that HMAC is indistinguishable from a function returning random bits.

Indistinguishability from random is a property of cryptographic output that says that there is no effective strategy by which an attacker that is viewing the output of the cryptographic function and a true random output can distinguish the two, where an "effective" strategy is one that has a non-negligible chance of success given bounded computing resources. If the output of a cryptographic function is indistinguishable from random, that implies that no information can be learned about the inputs of that function by examining the outputs.

We prove that our Cryptol HMAC specification has this indistinguishability property using an operational semantics of Cryptol we developed in Coq. The semantics enable us to reuse portions of the proof by Beringer et. al [6], which uses the Coq Foundational Cryptography Framework (FCF) library [20] to establish the security of the HMAC construction. We construct a Coq proof showing that our Cryptol specification is equivalent (when interpreted using the formal operational semantics) to the specification considered in the Beringer et. al work. The Cryptol specification is a stepping stone to automated verification of the s2n implementations, so when combined with the verification work we describe subsequently, we eventually establish that the implementation of HMAC in s2n also has the desired security property. The Coq code directly relating to HMAC is all on the s2n GitHub page. These proofs are not run as part of continuous integration, rather, they are only rerun in the unlikely event that the monolithic specification changes.

2.3 Low-level Specification

The formal specification of HMAC presented in the FIPS standard operates on a single *complete* message. However, network communication often requires the incremental processing of messages. Thus all modern implementations of HMAC provide an incremental interface with the following abstract types:

```
init : Key -> State
update : Message -> State -> State
digest : State -> MAC
```

The `init` function creates a state from a key, the `update` function updates that state incrementally with chunks of the message, and the `digest` function finalizes the state, producing the MAC.

The one-line monolithic specification is related to these incremental functions as follows. If we can partition a message m into $m = m_1 \| m_2 \| \ldots \| m_n$ then (in pseudo code/logic notation)

$$\mathsf{HMAC}(k, m) = \mathsf{digest}(\mathsf{update}(m_n(\ldots(\mathsf{update}\ m_1(\mathsf{init}\ k)))) \tag{1}$$

In other words, any MAC generated by partitioning a message and incrementally sending it in order through these functions should be equal to a MAC generated by the complete message HMAC interface used in the specification.

We prove that the incremental interface to HMAC is equivalent to the non-incremental version using a combination of manual proof in Coq and automated proof in Cryptol. Note that this equivalence property can be stated in an implementation-independent manner and proved outside of a program verification context. This is the approach we take—independently proving that the incremental and monolithic message interfaces compute the same HMAC, and then separately showing that s2n correctly implements the incremental interface.

Our Coq proof proceeds via induction over the number of partitions with the following lemmas establishing the relationship between the monolithic and iterative implementations. These lemmas are introduced as axioms in the Coq proof, but subsequently checked using SAW.

```
update_empty : forall s, HMAC_update empty_string s = s.

equiv_one : forall m k,
 HMAC_digest (HMAC_update m (HMAC_init k)) = HMAC k m.

update_concat : forall m1 m2 s,
 HMAC_update (concat m1 m2) s = HMAC_update m2 (HMAC_update m1 s).
```

The first lemma states that processing an empty message does not change the state. The second lemma states that applying the incremental interface to a single message is equivalent to applying the monolithic interface. These lemmas constitute the base cases for an inductive proof of equation (1) above. The last lemma states that calling update twice (first with `m1` and then with `m2`) results in the same state as calling update once with `m1` concatenated with `m2`. This constitutes the inductive step in the proof of (1).

The `update_empty` lemma can be proved by analyzing the code with symbolic values provided for the state s, as the state is of fixed size. The `equiv_one` and `update_concat` lemmas require reasoning about unbounded data. SAW has limited support for such proofs. In particular, it has support for equational rewriting of terms in its intermediate language, but not for induction. In the case of the `update_concat` lemma, a few simple builtin rewrite rules are sufficient to establish the statement for all message sizes. For `equiv_one`, a proof of the statement for all message sizes would require induction. Since SAW does not support induction, we prove that this statement holds for a finite number of key and message sizes. In theory we could still obtain a complete proof by checking all message sizes up to 16k bytes (the maximum size message permitted by the TLS standard). This may be tractable in a one-off proof, but for our continuously-applied

proofs we instead consider a smaller set of samples, chosen to cover all branches in the code. This yields a result that is short of full proof, but still provides much higher state space coverage than testing methods.

Given the three lemmas above, we then use Coq to prove the following theorem by induction on the list of partitions, `ms`.

```
HMAC key (fold_right concat empty_string ms) =
   HMAC_digest (fold_left (fun (st: state) msg =>
                                HMAC_update msg st)
                           ms
                          (HMAC_init key)).
```

The theorem establishes the equivalence of the incremental and monolithic interfaces for any decomposition of a message into any number of fragments of any size.

2.4 Implementation Verification

The incremental Cryptol specification is low-level enough that we were able to connect it to the s2n HMAC implementation using automated proof techniques. As this is the aspect of the verification effort that is critical for integration into an active development environment, we go into some detail, first discussing the tools that were used and then describing the structure of the proof.

Tools We use the Software Analysis Workbench (SAW) to orchestrate this step of the proof. SAW is effective both for manipulating the kinds of functional terms that arise from Cryptol, and for constructing functional models from imperative programs. It can be used to show equivalence of distinct software implementations (*e.g.* an implementation in C and one in Java) or equivalence of an implementation and an executable specification.

SAW uses bounded symbolic execution to translate Cryptol, Java, and C programs into logical expressions, and proves properties about the logical expressions using a combination of rewriting, SAT, and SMT. The result of the bounded symbolic execution of the input programs is a pure functional term representing the function's entire semantics. These extracted semantics are then related to the Cryptol specifications by way of precondition and postcondition assertions on the program state.

The top-level theorems we prove have some variables that are universally quantified (e.g. the key used in HMAC) and others that are parameters we instantiate to a constant (e.g. the size of the key). We achieve coverage for the latter by running the proof for several parameter instantiations. In some cases this is sufficient to cover all cases (e.g. the standard allows only a small finite number of key sizes). In others, the space of possible instantiations is large enough that fully covering it would yield runtimes too long to fit into the developer workflow (for example, messages can be up to 16k long). In such cases, we consider a smaller set of samples, chosen to cover all branches in the code.

This yields a result that is short of full proof, but still provides much higher state space coverage than testing methods.

Internally SAW reasons about C programs by first translating them to LLVM. For the remainder of the paper we will talk about the C code, although from a soundness perspective the C code must be compiled through LLVM for the proofs to apply to the compiled code.

Proof Structure The functions in the low-level Cryptol specification described above share the incremental format of the C program, and also consume arguments and operate on state that matches the usage of arguments and state in the C code. However, the Cryptol specification does not capture the layout of state in memory. This separates concerns and allows us to reason about equivalence of the monolithic and incremental interfaces in a more tractable purely functional setting, while performing the implementation proof in a context in which the specification and implementation are already structurally quite similar.

As an example of this structural similarity, the C function has type:

```
int s2n_hmac_update(struct s2n_hmac_state *state,
                    const void *in, uint32_t size);
```

We define a corresponding Cryptol specification with type:

```
hmac_update : {Size} (32 >= width Size) =>
              HMAC_state -> [Size][8] -> HMAC_state
```

These type signatures look a bit different, but they represent the same thing. In Cryptol, we list `Size` first, because it is a type, not a value. This means that we do not need to independently check that the input buffer (in Cryptol represented by the type `[Size][8]`) matches the size input—the Cryptol type system guarantees it. The type system also sets the constraint that the size doesn't exceed 2^{32}, a constraint set by the C type of `Size`.

We use SAW's SAWScript language to describe the expected memory layout of the C program, and to map the inputs and outputs of the Cryptol function to the inputs and outputs of the C program. The following code presents the SAWScript for the `hmac_update_spec` function.

```
let hmac_update_spec msg_size cfg = do {
    (msg_val, msg_pointer) <- ptr_to_fresh_array msg_size i8;
    (initial_state, state_pointer) <- setup_hmac_state cfg
    hmac_invariants initial_state cfg;

    execute_func [state_pointer, message_pointer, msg_size];

    let final_state =
        {{ hmac_update_c_state initial_state msg_val }};
    check_hmac_state state_pointer final_state;
    hmac_invariants final_state cfg;
    check_return zero;
};
```

This SAWScript code represents a Hoare triple, with the precondition and post condition separated by the body (the `execute_func` command), which performs the symbolic execution of the LLVM code using the provided arguments. Lines 2 and 3 are effectively universal quantification over the triple, setting up the values and pointers that match the type needed by the C function. The values `msg_val` and `initial_state` are referenced in both the C code and the Cryptol specification, whereas the pointers exist only on the C side.

Lines 8-10 capture that the final state resulting from executing the C function should be equivalent to the state produced by evaluating the Cryptol specification. Specifically, Lines 8 and 9 capture the output of the Cryptol specification (double curly braces denote Cryptol expressions within SAWScript) and Line 10 asserts that this state matches the C state present in memory at `state_pointer`. This is what ultimately establishes equivalence of the implementation and specification.

The proof is aided by maintaining a collection of state invariants, which are assumed to hold in Line 4 and are re-established in Line 11. These are manual invariants, but they occur as function specifications rather than appearing internal to loops. They only require modification in the event that the meaning of the HMAC state changes.

The `msg_size` parameter indicates how large of a message this particular proof should cover. Because SAW performs a bounded unrolling of the program under analysis, each proof must cover one fixed size for each unbounded data structure or iterative construct. However, by parameterizing the proof, it can easily be repeated for multiple sizes. Furthermore, as described in Section 2.3, we also prove in Coq that calling update twice with messages m_1 and m_2 is equivalent to calling it once with m_1 concatenated with m_2. As a consequence, the fixed size proofs we perform of update can be composed to guarantee that the update function is correct even over longer messages.

The `cfg` parameter contains configuration values for each of the six hashes that can be used with HMAC. The configuration values of interest to HMAC are the input and output sizes of the hash block function.

Given the specification of the C function above, we can now verify that the implementation satisfies the specification:

```
verify m "s2n_hmac_update"
  hash_ovs true (hmac_update_spec msg_size cfg) yices_hash_unint;
```

The `"s2n_hmac_update"` argument specifies the C function that we are verifying. `hash_ovs` is a list, defined elsewhere, that contains all of the *overrides* that the verification will use. An override is a specification that will be used in place of a particular implementation function and corresponds to what other tools call *stubs* or *models*. In this case, we've overridden all of the C hash functions, stating assumptions regarding their use of memory and their equivalence to Cryptol implementations of the same hash functions. When the verifier comes across a call to one of these hash functions in the C code, it will instead use the provided specification. The result is that our proof *assumes correct implementation of the hash functions*.

The fact that the structure of the low-level Cryptol specification matches the structure of the C code, coupled with SAW's use of SMT as the primary mechanism for discharging verification conditions, enables a proof that continues to work through a variety of code changes. In particular, changes to the code in function bodies often requires no corresponding specification or proof script change. Similarly, changes that add fields or change aspects of in-memory data structures that are not referenced by the specification do not require proof updates. Changes in the API (e.g. function arguments) do require proof script changes, but these are typically minor. Fixing a broken proof typically involves adding a new state field to the SAW script, updating the Cryptol specification to use that field correctly, and then passing the value of that field into the Cryptol program in the post-condition. If the Cryptol specification is incorrect, SAW will generate counterexamples that can be used to trace through the code and the spec together in order to discover the mismatch.

2.5 Integrating the Proof into Development

Integration with the s2n CI system mostly took place within the Travis configuration file for s2n. At the time of integration, targets for the build, integration testing, and fuzzing on both Linux and OSX already existed. We updated the Travis system with Bash scripts that automatically download and install the appropriate builds of SAW, Z3, and Yices into the Travis system. These files are in the s2n repository and can be reused by anyone under the Apache 2.0 license.

A Travis CI build can occur on any number of virtual machines, and each virtual machine is given an hour to complete. We run our HMAC proofs on configurations for six different hashes. For each of these configurations we check at three key-sizes in order to test the relevant cases in the implementation (small keys get padded, exact keys remain unchanged, and large keys are hashed). For each of those key-sizes we check six different message sizes. These proofs run in an average of ten minutes. We discovered that it's best to stay well clear of the 60 minute limit imposed by Travis in order to avoid false-negatives due to variations in execution time.

The proof runs alongside the tests that are present in the s2n repository on every build, and if the proof fails a flag is raised just as if a test case were to fail.

3 Proof of TLS Handshake

In addition to the HMAC and DRBG proofs, we have proved correctness of the TLS state machine implemented in s2n. Specifically, we have proved that (1) it implements a subset of TLS 1.2 as defined in IETF RFCs 5246 [21], 5077 [15] and 6066 [13] and (2) the socket corking API, which optimizes how data is split into packets, is used correctly. Formally, we proved that the implementation *refines* a specification (conversely, the specification *simulates* the implementation). We obtained this Cryptol specification, called the *RFC specification* by examining the RFCs and hand-compiling them into a Cryptol file complete with relevant

excerpts from the RFCs. We assume that the TLS handshake as specified in the RFCs is secure, and do not formalize nor verify any cryptographic properties of the specification. In the future, we would like to take a similar approach to that described in Section 2.2 to link our refinement proof with a specification-level security proof for TLS, such as that from miTLS [9].

The s2n state machine is designed to ensure correctness and security, preventing join-of-state-machines vulnerabilities like SMACK [7]. In addition, s2n allows increased throughput via the use of TCP socket corking, which combines several TLS records into one TCP frame where appropriate.

The states and transitions of the s2n state machine are encoded explicitly as linearized arrays, as opposed to being intertwined with message parsing and other logic. This is an elegant decomposition of the problem that makes most of the assumptions explicit and enables the use of common logic for message and error handling as well as protocol tracking.

Even with the carefully designed state machine implementation, formal specification and verification helped uncover a bug [10].

Structure of the TLS handshake state machine correctness proof. The automated proof of correctness of the TLS state machine has two parts (figure 2). First we establish an equivalence between the two functions[6] that drive the TLS handshake state machine in `s2n` and their respective specifications in Cryptol. Again we utilize *low-level* specifications that closely mirror the shape of the C functions. Our end goal, however, is correctness with respect to the standards, encoded in the *RFC specification* in Cryptol. The library implements only a subset of the standards, thus we can only prove a simulation relation and not equivalence. Namely, we show that every sequence of messages generated by the low-level specification starting from a valid initial state can be generated by the RFC specification starting from a related state. The dashed line in figure 2 shows at which points the states match at the implementation and specification levels.

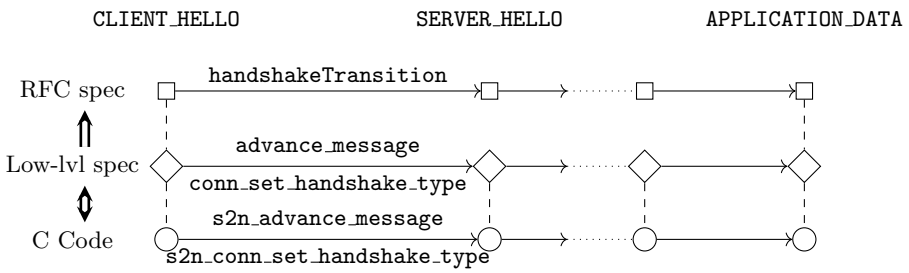

Fig. 2. Structure of the TLS handshake correctness proof

[6] `s2n_conn_set_handshake_type` and `s2n_advance_message`

RFC-based specification of the TLS handshake. The high-level handshake protocol specification that captures the TLS state machine is implemented in Cryptol and accounts for the protocol, message type and direction, as well as conditions for branching in terms of abstract connection parameters, but not message contents.

We represent the set of states as unsigned 5-bit integers (Listing 1). The state transition relation is represented by a Cryptol function `handshakeTransition` (Listing 2) which, given abstract connection parameters (Listing 3) and the current state returns the next state. If there is no valid next state, the state machine stutters. The parameters determine the transition to take in each state and represent configurations of the end-points as well as contents of the `HELLO` message sent by the other party. We kept the latter separate from the message specifications in order to avoid reasoning about message structure and parsing. We can still relate the abstract parameters to the implementation because they are captured in the connection state. Finally, the `message` function (Listing 4) gives the message type, protocol and direction for every state.

```
type State = [5]
(helloRequestSent : State) = 0
(clientHelloSent : State) = 1
(serverHelloSent : State) = 2
// ...
(serverCertificateStatusSent : State) = 23
```

Listing 1: Specification of TLS handshake protocol states

```
handshakeTransition : Parameters -> State -> State
handshakeTransition params old =
  snd (find fst (True, old) [ (old == from /\ p, to)
                            | (from, p, to) <- valid_transitions]) where
  valid_transitions =
    [(helloRequestSent, True, clientHelloSent)
    ,(clientHelloSent,  True, serverHelloSent)
    ,(serverHelloSent, params.keyExchange != DH_anon
                  /\ ~params.sessionTicket, serverCertificateSent)
    // ...
    ,(serverCertificateStatusSent, ~(keyExchangeNonEphemeral params)
      , serverKeyExchangeSent)
    ]
```

Listing 2: Specification of the TLS handshake state transition function. Valid transitions are encoded as triples $(start, transition\ condition, end)$.

```
type KeyExchange = [3]
(DH_anon : KeyExchange) = 0
// ...
(DH_RSA  : KeyExchange) = 5

type Parameters =
{keyExchange : KeyExchange   // Negotiated key exchange algorithm
,sessionTicket : Bit         // The client had a session ticket
,renewSessionTicket : Bit    // Server decides to renew a session ticket
,sendCertificateStatus : Bit // Server decides to send the certificate
                             //   status message
,requestClientCert : Bit     // Server requests a cert from the client
,includeSessionTicket : Bit} // Server includes a session ticket
                             //   extension in SERVER_HELLO
```

Listing 3: Abstract connection parameters

```
message : State -> Message
message = lookupDefault messages (mkMessage noSender data error)
  where messages =
    [(helloRequestSent, mkMessage server handshake helloRequest)
    ,(clientHelloSent, mkMessage client handshake clientHello)
    ,(serverHelloSent, mkMessage server handshake serverHello)
    // ...
    ,(serverChangeCipherSpecSent,
      mkMessage server changeCipherSpec changeCipherSpecMessage)
    ,(serverFinishedSent, mkMessage server handshake finished)
    ,(applicationDataTransmission, mkMessage both data applicationData)
    ]
```

Listing 4: Expected message sent/received in each handshake state

Socket corking. Socket corking is a mechanism for reducing packet fragmentation and increasing throughput by making sure full TCP frames are sent whenever possible. It is implemented in Linux and FreeBSD using the `TCP_CORK` and `TCP_NOPUSH` flags respectively. When the flag is set, the socket is considered corked, and the operating system will only send complete (filled up to the buffer length) TCP frames. When the flag is unset, the current buffer, as well as all future writes, are sent immediately.

Writing to an uncorked socket is possible, but undesirable as it might result in partial packets being sent, potentially reducing throughput. On the other hand, forgetting to uncork a socket after the last write can have more serious consequences. According to the documentation, Linux limits the duration of corking to 200 ms, while FreeBSD has no limit. Hence leaving a socket corked in FreeBSD might result in the data not being sent. We have verified that sockets are not corked or uncorked twice in a row. In addition, the structure of the message handling implementation in `s2n` helps us informally establish a stronger

corking safety property. Because explicit handshake message sequences include the direction the message is sent, we can establish that the socket is (un)corked appropriately when the message direction changes. In future work we plan to expand the scope of our proof to allow us to formally establish full corking safety.

4 Operationalizing the proof

We have integrated the checking of our proof into the build system of s2n, as well as the Continuous Integration (CI) system used to check the validity of code as it is added to the s2n repository on GitHub. For the green "build passed" badge displayed on the s2n GitHub page to appear, all code updates now must successfully verify with our proof scripts. Not only do the these checks run on committed code, they are also automatically run on all pull requests to the project. This allows the maintainers of s2n to quickly determine the correctness of submitted changes when they touch the code that we have proved. In this section we discuss aspects of our tooling that were important enablers of this integration.

Proof Robustness For this integration to work, our proofs must be robust in the face of code change. Evolving projects like s2n should not be slowed down by the need to update proofs every time the code changes. Too many proof updates can lead to significantly slowed development or, in the extreme case, to proofs being disabled or ignored in the CI environment. The automated nature of our proofs mean that they generally need to be changed only in the event of interface modifications—either to function declarations or state definitions.

Of these two, state changes are the most common, and can be quite complex considering that there are usually large possibly nested C structs involved (for example, the `s2n_connection` struct has around 50 fields, some of which are structs themselves). To avoid the developer pain that would arise if such struct updates caused the proof the break, we have structured the verification so that proof scripts do not require updates when the modified portions of the state do not affect the computation being proved. Recall that our proofs are focused on functional correctness. Thus in order to affect the proof, a new or modified field must influence the computation. Many struct changes target non-security-critical portions of the code (*e.g.* to track additional data for logging) and so do not meet this criterion. For such fields we prove that they are handled in a memory safe manner and that they do not affect the computation being performed by the code the proof script targets.

In the future, we intend to add the option to perform a "strict" version of this state handling logic to SAW, which would ensure that newly added fields are not modified at all by the portion of the code being proved. Such a check would ensure that the computation being analyzed computes the specified function *and nothing else* and would highlight cases in which new fields introduce undesirable data flows (e.g. incorrectly storing sensitive data). However even such an option

would not replace whole program data flow analysis, which we recommend in cases where there is concern about potential incorrect data handling.

Negative Test Cases Each of our proofs also includes a series of negative test cases as evidence that the tools are functioning properly. These test cases patch the code with a variety of mistakes that might actually occur and then run the same proof scripts using the same build tools to check that the tool detects the introduced error.

Examples of the negative test cases we use include an incorrect modification to a side-channel mitigation, running our TLS proofs on a version of the code with an extra call to cork and uncork, a version modified to allow early CCS, as well as a version with the incomplete handshake bug that we discovered in the process of developing the proof. Such tests are critical, both to display the value of the proofs, by providing them with realistic bugs to catch, and as a defense against possible bugs in the tool that may be introduced as it is updated.

Proof Metrics We also report real-time proof metrics. Our proof scripts print out JSON encoded statistics into the Travis logs. From there, we have developed an in-browser tool that scrapes the Travis logs for the project, compiling the relevant statistics into easily consumable charts and tables. The primary metrics we track are: 1) the number of lines of code that are analyzed by the proof (which increases as we develop proofs for more components of s2n), and 2) the number of times the verified code has been changed and re-analyzed (which tracks the ongoing value of the proof). This allows developers to easily track the impact of the proofs over time.

Since deployment of the proof to the CI system in November of 2016 our proofs have been re-played 956 times. This number does not account for proof replays performed in forks of the repository. We have had to update the proof four times. In all cases the proof update was complete before the code review process finished. Not all of these runs involved modification to the code that our proofs were about, however each of the runs increased the confidence of the maintainers in the relevant code changes, and each run reestablishes the correctness of the code to the public, who may not be aware of what code changed at each commit.

HMAC and DRBG each took roughly 3 man-months of engineering effort. The TLS handshake verification took longer at 8 months, though some of that time involved developing tool extensions to support reasoning about protocols. At the start of each project, the proof-writers were familiar with the proof tools but not with the algorithms or the s2n implementations of them. The effort amounts listed above include understanding the C code, writing the specifications in Cryptol, developing the code-spec proofs using SAW, the CI implementation work, and the process of merging the proof artifacts into the upstream code-base.

5 Conclusion

In this case study we have described the development and operation in practice of a continuously checked proof ensuring key properties of the TLS implementation used by many Amazon and AWS services. Based on several previous attempts to prove the correctness of s2n that either required too much developer interaction during modifications or where symbolic reasoning tools did not scale, we developed a proof structure that nearly eliminates the need for developers to understand or modify the proof following modifications to the code.

References

1. J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub. Jasmin: High-assurance and high-speed cryptography. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1807–1823. ACM, 2017.
2. Amazon.com, Inc. Amazon Simple Storage Service (s3). `https://aws.amazon.com/s3/`.
3. Amazon.com, Inc. s2n. `https://github.com/awslabs/s2n`. Accessed December 2017.
4. awslabs / s2n - Travis CI. `https://travis-ci.org/awslabs/s2n`.
5. M. Barbosa, D. Castro, and P. F. Silva. Compiling CAO: From cryptographic specifications to C implementations. In M. Abadi and S. Kremer, editors, *Principles of Security and Trust*, pages 240–244, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
6. L. Beringer, A. Petcher, Q. Y. Katherine, and A. W. Appel. Verified correctness and security of OpenSSL HMAC. In *USENIX Security Symposium*, page 207–221, 2015.
7. B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 535–552. IEEE, 2015.
8. K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. pages 445–459. IEEE, May 2013.
9. K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Bguelin. Proving the TLS handshake secure (as it is). Cryptology ePrint Archive, Report 2014/182, 2014. `https://eprint.iacr.org/2014/182`.
10. A. Chudnov. Missing branches in the handshake state machine. `https://github.com/awslabs/s2n/pull/551`, July 2017.
11. J. de Ruiter and E. Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., 2015. USENIX Association.
12. A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoué. Implementing and proving the TLS 1.3 record layer. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 463–482. IEEE, 2017.

13. D. E. r. Eastlake. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, Jan. 2011.
14. L. Erkök and J. Matthews. Pragmatic equivalence and safety checking in Cryptol. page 73. ACM Press, 2008.
15. P. Eronen, H. Tschofenig, H. Zhou, and J. A. Salowey. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077, Jan. 2008.
16. Galois, Inc. The software analysis workbench. `https://saw.galois.com/index.html`.
17. M. Gorelli. Deductive verification of the s2n HMAC code. Master's thesis, Universtity of Oxford, 2016.
18. D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. 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, Washington, D.C., 2015. USENIX Association.
19. P. O'Hearn. Continuous reasoning: Scaling the impact of formal methods. In *Logic in Computer Science (LICS)*, 2018.
20. A. Petcher and G. Morrisett. The Foundational Cryptography Framework. *arXiv:1410.3735 [cs]*, Oct. 2014. arXiv: 1410.3735.
21. E. Rescorla and T. Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Aug. 2008.
22. The Coq Development Team. The Coq proof assistant, version 8.7.1, Dec. 2017.
23. Trustinsoft. PolarSSL verification kit. `https://trust-in-soft.com/polarssl-verification-kit/`.
24. J. M. Turner. The keyed-Hash Message Authentication Code (HMAC). *Federal Information Processing Standards Publication*, 2008.
25. J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.