Verified Cryptographic Code for Everybody

Brett Boston[1], Samuel Breese[1], Joey Dodds[1], Mike Dodds[1], Brian Huffman[1], Adam Petcher[2], and Andrei Stefanescu[1]

[1] Galois, Inc.
[2] Amazon Web Services

Abstract. We have completed machine-assisted proofs of two highly-optimized cryptographic primitives, AES-256-GCM and SHA-384. We have verified that the implementations of these primitives, written in a mix of C and x86 assembly, are memory safe and functionally correct, by which we mean input-output equivalent to their algorithmic specifications. Our proofs were completed using SAW, a bounded cryptographic verification tool which we have extended to handle embedded x86. The code we have verified comes from AWS LibCrypto. This code is identical to BoringSSL and very similar to OpenSSL, from which it ultimately derives. We believe we are the first to formally verify these implementations, which protect the security of nearly everybody on the internet.

Keywords: Cryptography · Automated reasoning · Verification

1 Introduction

Widely-used cryptographic libraries such as OpenSSL [20], BoringSSL [16], and AWS LibCrypto [2] are an enticing target for formal verification. These libraries are used, to a first approximation, by everybody—or at least the four billion or so worldwide users of the internet. Each primitive in these libraries typically consists of a modest amount of code, but these primitives loom large in both their security and performance impact. Cryptographic primitives are also unusual in that they have clearly defined specifications and very few dependencies, which removes some major challenges from general-purpose verification. As a result, in recent years many efforts have been made to verify cryptographic library code.

However, despite significant progress, widely-used cryptographic libraries have resisted verification, at least for the versions of the primitives that are used in practice. This is because these primitives are some of the most heavily optimized pieces of code in existence. For a cloud service, every packet involves a call to at least one cryptographic primitive, so even small optimizations will have large performance and cost impacts. As a result, for AES and SHA there is an enormous gap in complexity between simple and easily verified high-level reference implementations, and the highly optimized implementations used in production.

Optimizations create several difficulties when verifying cryptographic primitives. First, primitives are typically written in a mix of C and assembly. This means that a verification tool must model both of these languages and the manner in which they can interact. Furthermore, each optimization step inherently increases the difficulty of verification, because each requires one or more theorems showing that the optimization is sound. To add to this, many of these optimizations break the abstractions used in algorithm specifications. For example, the SHA-384 specification is defined using a function called `Sigma0` that is unfolded and rearranged during the optimisation process (see subsection 6.1). Solver-based automation typically struggles to recover these abstractions.

The verification of cryptographic code has seen huge advances in recent years. Purpose-built libraries such as EverCrypt [21] can now match the performance of hand-tuned OpenSSL. These correct-by-construction libraries may be the future, but as of 2021 they have not yet seen wide mainstream adoption. Our aim as formal methods practitioners is to verify the cryptographic code on which users depend. What has been missing until now is the ability to verify the legacy cryptographic code that runs in production for hundreds of millions of users. This is the problem we solve.

Approach and Results. We have formally verified the memory safety and functional correctness of two key cryptographic primitives, AES-256-GCM and SHA-384 as they currently appear in the new AWS LibCrypto library (AWS-LC) [2]. AWS-LC is a general-purpose library maintained by Amazon Web Services for use with AWS applications. We targeted these algorithms in particular because they are used within AWS and included in the Commercial National Security Algorithms Suite [18]. We chose a block cipher and a hashing algorithm in order to cover multiple algorithm types and to be representative of other algorithms in AWS-LC.

Cryptographic algorithms have fixed specifications which permit a narrow range of designs, and as a result, implementations change slowly. The AES-256-GCM and SHA-384 implementations in AWS-LC are identical to those in Google's BoringSSL library, and as a result, our proofs apply to it as well. For these primitives, there are only small differences between BoringSSL and OpenSSL, and we are confident our proofs would also apply to OpenSSL with minor modifications.

Our proofs show that the implementations of AES-256-GCM and SHA-384 are input-output equivalent to formal specifications of their expected behaviour. We write our specifications in Cryptol [11], a pre-existing high-level language designed for use by cryptographic experts. Cryptol specifications are executable, so our proofs establish that for any input, the implementation and specification produce exactly the same result. To boot, our proofs guarantee that the code is free of undefined behaviour such as memory safety errors, meaning that any remaining correctness errors are local to the code being proved and cannot affect the calling context. We do not verify side-channel properties, nor do we analyse cryptographic security properties of the AES-256-GCM and SHA-384 algorithms.

We performed these proofs using the Software Analysis Workbench (SAW) [14]. SAW is an industrial verification tool designed to prove equivalence properties between abstract specifications and lower-level, more optimized implementations. SAW is a bounded verifier: loops must be verified under preconditions that guarantee termination, and data-structures must be statically allocated with bounded sizes.

We have run our proofs on fixed sizes of input data, i.e. fixed numbers of bytes to be hashed / encrypted / decrypted. The number of loop iterations in these algorithms are strictly fixed by the input size so this also implicitly bounds the execution length. We chose these sizes so as to exercise all branches and boundary conditions in the code and specification (in this, we follow Galois and AWS's previous work: see Chudnov *et al.* [7]). We discuss the scope and limitations of our proof in section 7.

Each proof of a cryptographic primitive in SAW has two stages. In the first, the imperative input code is converted to a functional term using bounded symbolic execution. This depends on a high-fidelity model of the input languages. SAW already had an LLVM model used for C and C++ verification. For AES-256-GCM and SHA-384 we developed a new SAW model of x86 assembly, along with an interface with SAW's existing LLVM model. As well as modeling core x86, this also included modeling special-purpose instructions used to achieve high performance. A successful conversion only occurs for well-defined programs, and implies that the program is free of undefined behavior under the given preconditions.

In the second stage of a SAW proof, the symbolic term is compared to a specification term written in Cryptol. For many applications, SAW can discharge these equivalences automatically, but this is where the optimizations in AES-256-GCM and SHA-384 made verification much more challenging. The proof steps involved cannot be discharged automatically by current solvers, so instead, our proofs make careful use of rewriting logic to massage the terms into a form that can be discharged. Some of these proof steps may be amenable to automated solving in future.

Our proofs were developed collaboratively between a team of expert verification engineers. As well as technical innovation, these proofs also required careful proof engineering. By this, we mean the analog of software engineering—a combination of proof design, tool design, and team working practices which makes it possible to execute effectively on a verification goal. We found that to a degree, proof engineering *is* software engineering; that is, successful proof engineering has similarities to the practices needed when developing a challenging software project.

Aside from proofs and tool capabilities, there is something else notable about our project: we verify code that was never intended for formal verification. This is in contrast to many other efforts, which target systems that were designed with assurance in mind. For example, Galois and AWS previously verified an Amazon TLS library that was purpose-built as a high-assurance alternative to OpenSSL's TLS support [7], while in the EverCrypt library, code and proof were developed in parallel, and even the API was designed to simplify specifications [21]. We verify legacy code because this is the code that is actually used in AWS-LC and its predecessors.

Contributions. The key contributions of this paper are as follows:

- Proofs of correctness for highly optimized versions of AES-256-GCM and SHA-384, as they appear in AWS LibCrypto and BoringSSL.
- A verifier for mixed C and x86 code which allows precise reasoning about functional correctness. This capability is built into the industrial verification tool, SAW.
- A simple system of rewrite tactics which is powerful enough to allow verification of highly optimised cryptographic algorithms.
- Lessons learned in proof engineering when applying an industry verification tool to a challenging piece of legacy cryptographic code.

All proof scripts are available online[3].

1.1 Related Work

There is a considerable amount of recent work in cryptographic verification, representing a large space of application domains and design requirements. While our work is widely applicable, we do not consider it a one-size-fits-all solution. We discuss how a developer might choose between the many verified cryptography efforts in subsection 7.2. Here we give an overview of projects that target C or x86, or that are closely related technically. We do not review work on verifying cryptographic security properties, which is orthogonal to the problem of verifying that code matches algorithm.

The closest work to ours in terms of technical approach is Galois and AWS's previous work verifying the HMAC and DRBG primitives in the AWS s2n TLS library [7]. Just as we do, they use SAW to verify production cryptographic code. The main difference from our current project is the complexity of the primitives verified. The HMAC and DRBG primitives are inherently simpler algorithms, and are written in C, rather than x86. Furthermore, this code was designed for verification, unlike the OpenSSL-derived code we target. In earlier work, Ye *et al.* also verified C versions of HMAC and DRBG from OpenSSL using the foundational Verified Software Toolchain (VST) [22].

The Everest project has developed verified C/x86 cryptographic library called EverCrypt [5,10,23,21]. Recent results are extremely impressive, with performance comparable to highly optimised OpenSSL code. However, EverCrypt represent a different philosophy from ours, where the library and proof are co-designed, and in some cases code is synthesized. This approach looks towards a future where such libraries replace hand-written libraries like AWS-LC, BoringSSL, and OpenSSL. Our philosophy is complementary: we verify code as it currently exists while we wait for the future to arrive.

EverCrypt also differs in that they use a proof-assistant style of reasoning more similar to Coq or Isabelle. The advantage of this is that proofs are very flexible—for example, they work for unbounded input sizes. However, the cost is that proofs are relatively more verbose. Proof size is hard to estimate in EverCrypt, because the proof and implementation are mixed, but the earlier Vale

[3] https://github.com/awslabs/aws-lc-verification

paper [10] suggests that EverCrypt's proof of AES-GCM uses 2000 lines of proof library plus additional proof mixed in. In comparison, SAW is designed to automate reasoning where possible, and the proof of AES-256-GCM implementation takes us less than 1000 lines of proof (including white-space and comments, for attempted apples-to-apples comparison).

The CASM [17] project verifies x86-based cryptography taken from OpenSSL, including SHA-256 (we verify SHA-384). CASM's toolchain is similar to ours, based on symbolic execution and SMT solvers. However, CASM only examines functions over message blocks, rather than the whole SHA-256 algorithm. CASM also does not verify the most highly optimised versions of this algorithm. For example, it omits x86 EVP and vector operations, two of the main challenges.

Fiat Crypto [9] is a related approach, although it does not apply to the algorithms proved in this paper. It foundationally generates portable C field arithmetic implementations from a high level specification. Code synthesized by Fiat Crypto has already been added to OpenSSL. Jasmin [1] is another foundational synthesis approach. It generates high-performance vectorized x86 implementations. The Jasmin implementation of ChaCha20-Poly1305 outperforms similar hand-optimized implementations. We have not seen Jasmin implementations of SHA-2 or AES-GCM.

SAW's approach has some similarities to model checking, in that it is a bounded verification technique. However, proofs are based on symbolic execution, that is, construction of logical terms representing the program denotation, and proofs are bounded on input buffer size, not program execution length per se.

2 Project Design Constraints

Our objective in this project was to verify the cryptographic code which is actually deployed, and to ensure it stays verified as it changes over time[4]. To do this, we used *continuous reasoning*, a term due to Peter O'Hearn [19]. In continuous reasoning, there is a tight connection between code, software engineering process, and verification tools. Several recent industry projects have successfully used continuous reasoning practices. It was also important that our tools maintain the existing institutional trust in the original codebase—this ruled out whole-code replacements such as EverCrypt. This resulted in the following design constraints:

- Proofs had to run on the executed code, rather than a model / abstraction. This was to minimize the trusted base, and ensure that our proofs stayed in sync with the code as it evolved.
- Proofs had to run automatically with a low enough time budget to integrate with continuous integration checking. This ensures that errors are detected at the time code is changing, which increases the probability of fixes.

[4] In fact, we do not expect AES-256-GCM and SHA-384 to change often in AWS-LC, but this work takes place in the context of a larger AWS-LC assurance project.

– Proofs had to avoid modifications to the original source code, and instead exist as separate supporting files. Our experience is that teams are typically very reluctant to modify original source code, even with non-functional annotations.
– The proof toolchain had to operate independently of the software build system. This was to avoid introducing untrusted tools into critical development pathways.

These constraints led us to use the SAW tool as our basis for verification [14]. Our project can be seen as a follow on to Galois and AWS's prior verification of AWS s2n which had many of the same design objectives [7]. Chudnov *et al.* showed that SAW can be used for continuous reasoning for a relatively simple piece of C cryptography. The difference in our current project is the inherent difficulty of verifying the code.

3 AES-256-GCM and SHA-384 Proof Structure

Conceptually, SAW's approach to proof works as follows. The tool symbolically executes C and x86 code, resulting in a collection of functional terms. A term describes every program output mathematically as a function of program inputs. Once side conditions have been discharged, completion of symbolic execution also implies that the program is safe: that is, memory safety errors cannot occur. In the final step of the proof, these functional terms are compared to specifications using a solver to determine whether they are equivalent.

Interfaces. At the top level of our proof, we verify the AWS LC primitives against OpenSSL's EVP interface[5]. OpenSSL and its descendants use this interface to make it easy to swap out algorithms without exposing their implementations. This complicates the verification task by hiding functions behind pointers and union types. It has also attempted to remain largely backwards compatible for years, resulting in an API that is not as clean as it might be otherwise. Perhaps for these reasons, previous cryptographic verification projects have not verified the EVP interface.

SAW-script specifications. The top-level EVP specifications are defined in SAW-script, the high-level control language for SAW. Figure 1 shows part of the SAW-script EVP interface for AES-256-GCM. In its form, this interface consists of a series of instructions in SAW-script, but in its effect, it is a Hoare-style pre/post specification. The interface sets up symbolic memory (the pre-condition), symbolically executes the function (`crucible_execute_func`), and then checks that the resulting symbolic memory contains the correct values (the post-condition).

For AES, the main purpose of the pre-condition is to define the layout of memory that results from the AES initialization function. Because we define post-condition for the initialization function that match the specification given

[5] https://wiki.openssl.org/index.php/EVP

```
let EVP_CipherUpdate_spec enc gcm_len len = do {
  // ... some cipher set-up omitted (5 lines)

  cipher_data_ptr <- crucible_alloc_aligned 16 (llvm_struct "struct.EVP_AES_GCM_CTX");
  points_to_EVP_AES_GCM_CTX cipher_data_ptr ctx mres {{ 1 : [32] }} 0xffffffff;

  ctx_ptr <- crucible_alloc_readonly (llvm_struct "struct.evp_cipher_ctx_st");
  points_to_evp_cipher_ctx_st ctx_ptr cipher_ptr cipher_data_ptr enc;

  (in_, in_ptr) <- ptr_to_fresh_readonly "in" (llvm_array len (llvm_int 8));
  out_ptr <- crucible_alloc (llvm_array len (llvm_int 8));
  out_len_ptr <- crucible_alloc (llvm_int 32);

  crucible_execute_func [ctx_ptr, out_ptr, out_len_ptr, in_ptr,
                         (crucible_term {{ `len : [32] }})];

  let ctx' = {{ cipher_update enc ctx in_ }};

  // ... some cipher invariants omitted (3 lines)

  crucible_points_to out_ptr (crucible_term {{ ctr32_encrypt ctx in_ }});
  crucible_points_to out_len_ptr (crucible_term {{ `len : [32] }});
  crucible_return (crucible_term {{ 1 : [32] }});
};
```

Fig. 1. Part of the EVP interface for AES-256-GCM.

here, we can end-to-end verify the common use case of initializing memory, encrypting some input, and returning the result.

The script defines the memory pre- and post-conditions for the function using points-to assertions. In SAW-script, we allocate symbolic memory at specific sizes using the `crucible_alloc` commands. We can then use the `points_to` command to specify that a pointer points to symbolic memory. The `ptr_to_fresh` command is a convenience function that allocates a pointer, and then initializes it with symbolic memory.

SAW's logic is less expressive than a full separation logic, but specifications can naturally be interpreted in terms of separation, including the property that memory cells do not overlap. To make the memory layout easier to understand, consider the following separation logic triple, which roughly corresponds to the layout defined in the SAW-script:

$$\{\text{cipher\_data\_ptr} \mapsto \text{ctx...} \ * \ \text{in\_ptr} \mapsto \mathit{in} \ * \ \text{out\_ptr} \mapsto (\_ : [\mathit{len}])\}$$

$$\texttt{EVP\_CipherUpdate(ctx\_ptr, out\_ptr, out\_len\_ptr, in\_ptr, len)}$$

$$\left\{ \begin{array}{c} \text{cipher\_data\_ptr} \mapsto \text{cipher\_update(ctx...)} \ * \\ \text{in\_ptr} \mapsto \mathit{in} \ * \ \text{out\_ptr} \mapsto \text{ctr32\_encrypt(ctx, in)} \end{array} \right\}$$

Rather than syntactically divide the pre-condition and post-condition, as in a Hoare triple, the two are divided by the call to `crucible_execute_func`, which indicates symbolic execution of the target C or x86 function. Crucible is the intermediate language for symbolic execution used by SAW. Internally, the semantics of LLVM, x86, and other SAW input languages are defined by translation to Crucible.

One reason for the complexity of these specification is that SAW differentiates between data that is allocated and initialized and data that is just initialized. Other verification tools tend to treat all allocated data as initialized (for example, this is true of CBMC [8]). This is generally a sound approximation because C compilers tend to behave predictably, but our approach is more accurate to the specification of C.

Functional specifications. The other role of SAW-script is to verify the connection between the implementation and algorithmic specification. In SAW, specification are written in Cryptol, a domain-specific language designed for cryptographic specifications [11]. In the postcondition of the script, we use references to Cryptol functions to map the outputs of running the program to the outputs of our specification programs, `ctr32_encrypt` and `cipher_update`. The final lines of the specification assert that the memory cells resulting from the program must match the required values, i.e. those that would result from executing the Cryptol specification.

We show `ctr32encrypt` in Figure 2. This function defines the top-level behavior of the CTR mode of encryption, which repeatedly increments an initialization vector, encrypts the incremented value with the secret key, and performs an XOR of that encryption with the plaintext.

The first line of the specification defines the type of the function, parameterized by type variable `n`. `AES_GCM_Ctx` is a structure used to maintain state for the incremental interface to AES, which allows for data to be encrypted and decrypted as it becomes available, rather than all at once. The `[n][8]` arguments are sequences of bytes with length `n`.

The function body consists of a sequence comprehension. This takes input bytes one at a time, and labels them with `i`, which draws from the sequence counting up from `ctx.len`. The separate function `EKij` performs the encryption step using the initialization vector and the key contained in the context. The `take` and `drop` functions are used to convert the 64-bit length contained in the context to a 32-bit number required by the `EKij` function.

Another example of a functional specification is the following line describing the `Sigma0` function:

```
S0 x = (x >>> 28) ^ (x >>> 34) ^ (x >>> 39)
```

In the SHA-384 code, this function is implemented by the Perl code given in Figure 3. This does not execute directly, but rather generates assembly code, which is what we verify. The instructions `ror` and `xor` correspond to the cryptol operations >>> and ^ respectively.

```
ctr32_encrypt : {n} (fin n) => AES_GCM_Ctx -> [n][8] -> [n][8]
ctr32_encrypt ctx in = out
  where
    out = [ byte ^ (EKij ctx ((take`{32} (drop`{28} i)) + 1) (drop`{60} i)) |
                         byte <- in | i <- [ctx.len ...] ]
```

Fig. 2. Top-level Cryptol specification for AES update.

```
        '&ror         ($a1,39-34)',
        '&xor         ($a1,$a)',
        '&ror         ($a1,34-28)',
        '&xor         ($a1,$a)',
        '&ror         ($a1,28)',                # Sigma0(a)
```

Fig. 3. Perl implementation of internal SHA computation.

In order to include the implementation here, some constants have been substituted, and we have extracted the relevant lines from around 20 other lines calculating other parts of SHA. Those lines are mixed in with even more lines of non-interfering SHA calculations, presumably in order to keep the processor saturated. Symbolic execution allows us to reason just about these lines of code, because interleaved instructions that don't change the result of the computation in a relevant way will not be included when reasoning about the results of individual computations.

Notice also that the shift amounts are different between the functional specification and the code. In Cryptol, the shift amounts are 28, 34, and 39, but in the implementation, we see shifts by $39 - 34$, $34 - 28$, and 28. This is a performance optimisation, but it makes the proof effort more difficult. To close this gap, we use a system of verified rewrites (see section 6).

Verification process. Once a specification has been defined, it must be verified. SAW divides verification into two phases: symbolic execution, and verification of equivalence. Symbolic execution converts an imperative operation into a functional term suitable for automated reasoning. Even without specifying the expected high-level behaviour of the AES function, the memory layout defined in the pre-condition is enough for symbolic execution to complete, which has the effect of proving the imperative code memory safe. We typically verify memory safety in this way before developing a specification. This lets us separate concerns between functional and safety properties.

The final task once symbolic term has been generated is to compare it to a specification term. SAW uses SMT solving to discharge these proofs, and in most use cases, these can be completed automatically. However, the complexity of the optimization stages in AES-256-GCM and SHA-384 makes the gap between specification and implementation too large to be completely automated. SAW solves this with a small tactic language embedded into SAW-script that supports term rewriting. Each of these rewrites is then verified by the solver. We discuss rewriting further in section 6.

Modular reasoning. Symbolic execution is a precise technique with hard limits on its scalability. The AES-256-GCM and SHA-384 functions are too large to be symbolically executed in their entirety. SAW solves this problem through using a modular reasoning system called *overrides*. SAW treats specifications as executable code that can be freely substituted for implementation functions. When a function is verified equivalent to a Cryptol specification, calls to that function can be overridden (i.e replaced) during symbolic execution. As Cryptol specifications are typically much less complex than implementations, this massively increases the tractability of the verification task.

As a result, a typical SAW proof consists of a hierarchy of equivalence proofs. The proof begins at the leaf functions, which are verified by symbolic execution. The functions at the next level are then symbolically executed with the leaf functions replaced by their specifications. These are then also added to the library of verified functions. This proceeds until the top-level function is verified. One of the main tasks when developing a SAW proof is defining these internal specifications (our proof is unusual in that we also needed a significant number of rewrite rules).

We also use the override mechanism at the interface between C and x86 code. Functions in x86 are proved equivalent to Cryptol specifications, and these specifications can then be used as overrides in the surrounding C context. This approach works because we have defined a compatible memory model that works for both C and x86 code—see section 5 for more.

Finally, the override functionality can be used to assume specifications for functionality that has been assumed, not verified. This is useful for library calls that might be out of scope for a particular project, but that might be verified in the future. For example, Chudnov *et al.*'s SAW proofs for s2n [7] use this approach to parameterize the proofs of HMAC and DRBG over different primitives[6]. In our proofs, the only assumptions we make are that that `OPENSSL_malloc` and `OPENSSL_free` behave correctly.

4 SAW's Verification Pipeline

SAW is structured as a pipeline of linked verification stages. The inputs to the pipeline are, firstly, executable mathematical specifications for the top-level function, and selected sub-functions; secondly, the compiled code, made up of LLVM and embedded x86 binary code; and thirdly, a proof script which sets up memory, identifies the mapping between Cryptol specifications and function interfaces, and contains the rewrites that are applied to the resulting logical terms. The verification pipeline then works as follows:

[6] In fact, we have now verified some of the primitives that were only assumed in this previous work, meaning it should be possible to stitch these proofs together end-to-end

1. The x86 binary is extracted from the LLVM and decompiled into a CFG representation that recovers the x86 instructions and control-flow structure. This relies on a SAW sibling project called Macaw [12].
2. The x86 control-flow graph and LLVM code are divided into functions at the interfaces identified in the SAW-script file.
3. Beginning at the leaves of the call-graph, each x86 and LLVM function is symbolically executed, resulting in a term written in a intermediate language called SAW-core. At this stage, any already-verified functions are substituted for Cryptol overrides.
4. If a function has an associated Cryptol specification, it too is symbolically executed, resulting in a specification term in SAW-core.
5. The function term and specification term are rewritten using the rewrites defined in SAW-script.
6. The rewritten function and specification terms are proved equivalent through a generic solver interface library called What4 [15].

Verification proceeds with functions progressively higher on the call-graph, until the top-level equivalence is proved between code and specification.

While the structure of this pipeline is simple, making it work for real code requires a significant amount of tool sophistication. SAW is the product of many years of refinement and development, and we used many of the components in this pipeline without modification.

Our C support is based on SAW's LLVM support, which is mature, and has been used in many other industry and government verification projects—for example, Chudnov *et al.* [7]. While we do not claim complete coverage of the standard, in practice we rarely need to add new C language features to SAW. Likewise, Cryptol support is built into SAW and is designed to be symbolically executed, so this part of the tool required no modifications. The Macaw and What4 tools similarly functioned without modification.

Therefore, in this paper we focus on the new capabilities of the tool: our symbolic execution of x86 instructions, and verified rewrites. For a more detailed treatment of the SAW suite as a whole, readers should look at the SAW documentation and tutorial [14,13].

5 New Capability: x86 Semantics

The first SAW capability we developed for this project was symbolic execution for x86 assembly code, including support for mixed C/x86 code. Doing this required us to solve two problems. First, decompiling the binary into a series of x86 instructions, and second, defining the semantics of instructions, which mainly involves defining the model of memory.

To decompile we use Macaw, a SAW sibling project which is able to parse Elf binaries and output a control-flow graph complete with the representation of the x86 instructions [12]. We treat Macaw as a black box, and in fact any decompiler with similar capabilities could serve in its place.

Once the CFG has been constructed, we apply our x86 semantics. For the behaviour of individual instructions, we consulted the Intel manual. We note that processor manuals contain errors, and hand-encoding the semantics could also introduce errors. However, we have reasonable confidence in this encoding because, in practice, most conceivable errors would immediately cause the proof to fail. This is because cryptographic functions are very sensitive to small changes: most small value errors would result in a dramatically different output.

Much more important and subtle is the memory model, which describes under what conditions reads and writes to memory can occur, as well as describing how reads and writes can be combined to store and retrieve values. Unlike C, there are almost no accepted memory usage rules for assembly programming, aside from the conventions used in a particular program and the Application Binary Interface (ABI) for functions that can be called externally. Fortunately, AES and SHA implementations are designed to be called by C programs. They therefore must follow C-like conventions and respect the ABI. Memory is used to get inputs and define outputs, read global constants, and maintain a stack for storing temporary results. Functions always respect the boundaries of data as provided. Because of this, we were able to adapt SAW's well-tested model used for LLVM support.

In SAW's memory model, addresses are represented by a pair of integers: the first integer is a base address, identifying an allocated memory region, while the second is an offset into the region. Memory operations, such as pointer arithmetic and pointer comparisons, are only well-defined for addresses in the same region.

Even after defining this model, we had to decide how to apply it within the proof. There were two options: (i) modeling the entire memory as a single region, and (ii) representing different objects as separate regions. The former is the more flexible because it does not enforce any invariants on the way that memory is used. Any read or write within the entire memory region is valid at any time. This comes at an increased cost of manually specifying necessary invariants. For example, each function would have to manually encode the memory region it might write to so that its calling function can predict all of the side effects of calling it.

Instead, we take the second approach: automatically specifying such memory invariants as part of the way that memory can be used. This means that some valid assembly will be impossible to verify. It could be completely safe and correct, but because it violates the strict memory model we've chosen, our tool will be unable to reason about it. On the other hand, the memory model we chose works for all of the cryptographic assembly code we've run into, and implementing the memory model in this way saves us a substantial amount of specification and proof work.

It is not surprising that this approach works; The models and abstractions C uses for memory are useful in assembly as well. Furthermore, the ABI and the C memory model have heavily co-evolved, making the C memory model a natural fit for assembly functions that match the ABI.

The memory model is applied by symbolic execution of the CFG that results from Macaw. This symbolic execution has two main functions: efficiently update a symbolic representation of memory, and discharge side conditions that must hold in order for symbolic execution to continue. The result is a SAW-core term representing the input-output behaviour of the x86 binary code.

6 New Capability: Verified Rewrites

The second SAW capability we developed was a simple language of term rewrites for use in proofs.

After symbolic terms have been constructed from C, x86, or Cryptol, we must prove equivalences between these terms. The design goal with SAW is that these proofs are completed mostly automatically using SMT solvers. While this has worked well in previous, less-complicated proofs, the functional terms that result from AES-256-GCM and SHA-384 often proved to be intractable for the solvers without preprocessing. This is exactly because these algorithms are so heavily optimised, as we have discussed above.

In order to solve this, we introduce a language of equivalences between terms that are themselves verified by the solver. By applying these rewrites, we can close the gap between the more abstract Cryptol term and the optimized C/x86 term. These rewrites serve as a small tactic language for controlling the proof, while preserving the principle that SAW proofs are mostly automatic.

To illustrate how this works, we consider an example rewrite from our SHA-384 proof. In the Cryptol portion of our proof, we define the following function, `S0` (shortened for convenience from `Sigma0`):

```
 S0 x = (x >>> 28) ^ (x >>> 34) ^ (x >>> 39)
```

In Cryptol, >>> and <<< are right and left rotation respectively, while ^ is XOR. In order to complete the proof, we need to be able to rewrite occurrences of this function. To do this, we define the following rewrite, `Sigma0_thm`:

```
Sigma0_thm <- prove_folding_theorem
                {{ \x -> (x ^ ((x ^ (x <<< 59)) <<< 58)) <<< 36 == S0 x }};
```

The left hand side of this equation is how symbolic execution interprets the code in Figure 3. The rotate-rights have been swapped to rotate-lefts, which allows our semantics to model both types of instruction by rotate-left. In order to swap the rotates, we subtract the rotate amount from 64, which is why we have a different set of constants than we see in either the specification or the implementation.

The solver verifies this equivalence for all possible values of `x` and saves it with the name `Sigma0_thm`. In this case, we verify the equality using the ABC solver [6] through What4, but different solvers can be applied as needed to provide different equivalences.

Consider the following SAWscript command, which verifies an x86 function matches its specification:

```
sha512_block_data_order_spec <-
  crucible_llvm_verify_x86 m "<filename>" "sha512_block_data_order"
    [ ("K512", 5120) ] // Initialize global for round constants
    true
    sha512_block_data_order_spec
    (do {
      simplify (cryptol_ss ());                       // std simplifications
      simplify (addsimps thms empty_ss);              // folding theorems
      simplify (addsimp concat_assoc_thm empty_ss);  // final theorem
      w4_unint_yices ["S0", "S1", "s0", "s1", "Ch"]; // uninterpreted fns
     });
```

Here, the do-block defines the order in which the simplification rewrite rules are applied. The folding theorems `thms` contains 30 rewrite rules, including the `Sigma0_thm` presented above. The `concat_assoc_thm` theorem normalizes the concatenations that result from other proof rules. The final line of this script instructs the Yices solver to treat certain functions as uninterpreted, including the `S0` function. This illustrates the usefulness of the rewriting support. Rather than reasoning about the `S0` function directly, we rely on the verified rewrites. This allows us to abstract away from complexity that previously made the proof infeasible for the solver.

Overall, the tactics we use for SAW proofs constitute a very simple decision procedure, made up almost exclusively of user-supplied rewrites. The other main mechanism we have for guiding proofs is the modular override system described in section 3, which allows us to decompose proof tasks into lemmas, at least at the granularity of functions. In practice, we have found that these capabilities are sufficient to meet the needs of proving cryptographic implementations correct with respect to specifications.

Ultimately, we may find ourselves limited by the tools available in SAW for controlling the proof process, particularly if we attempt to prove higher degrees of abstraction between specification and implementation. These more manual proofs largely fall outside of the scope of what SAW aims to do well. An ideal solution would be to export proof goals to Coq, Lean, or F*, all of which already have highly-usable better tools for manual proof. Chudnov *et al.* have previously demonstrated that SAW proofs about code and more abstract Coq proofs can be connected in this way.

6.1 Role of Rewrites in AES-256-GCM and SHA-384 Proofs

Rewrites in SAW can be seen as a small tactic language, serving a similar purpose to proof tactics in Coq or Isabelle. However, SAW occupies a very different point in design space, because it is designed to maximize proof automation. Heavy use of SMT-backed automation is the reason our proofs were feasible, but if the automation makes poor choices, it can also obstruct the proofs. We use rewrites along with appropriate choices of abstraction boundaries, to recover abstractions that automation would not discover itself.

For example, consider the `Sigma0_thm` rewrite defined above. The solver can verify the rewrite when supplied in isolation. However in the context of SHA, the solver fails to identify this as a valuable fact. One reason is that `S0` is a function that is present in the Cryptol specification, but this abstraction is lost when we symbolically execute an x86 function. The rewrites replace occurrences of `S0` with an uninterpreted function, pruning the proof space dramatically.

However, there is the trade-off in reintroducing such an abstraction. Even if the abstraction holds locally, the functionality that calls that abstraction might depend on the internal functionality. In that case, swapping out the code for an uninterpreted function could actually turn a solvable goal into an unsolvable one. The answer is to choose these rewrites carefully: this is one of the main intellectual challenges in completing a proof. In general, this problem is undecidable. For example the rewrite rules inferred may not terminate, this means that at best it might be a guided special-purpose mode of solvers, rather than a general purpose approach.

Our rewrites plug into SAW late in the pipeline, after many of SAW's optimizations. This means that rewrites sometimes have to compensate for earlier optimizations. SAW is designed to aggressively optimize terms into a form suitable for the solver, and in some cases, this means breaking up abstractions that would be useful in completing the proof. In these cases, our rewrites must operate on the post-optimization proof term.

For example, in one case, it would have been desirable for our proof to use the term:

```
{{ \x -> (slice_59_5_0  x) # (slice_0_59_5  x) == x <<< 59 }};
```

However, SAW discovered that it could drop off the operation on the final byte of `x`, but to do so, it had to break up `x` into its constituent bytes. This is a desirable optimization if the term is passed to the solver directly, because the solver itself will reason at the level of bytes. However, this made writing an appropriate rewrite for our proof much more challenging. We include the eventual rewrite rule in Figure 4. Again, a large amount of the intellectual challenge with our proof rested in finding appropriate rewrites that integrated with SAW's existing automation.

7 Results and Lessons Learned

Our proofs run on the current version of AWS-LC [2] as of January 2021, built using the default compiler flags. We verify the AVX implementation of SHA-384, which is the current fastest implementation. Our AES-256-GCM proof uses the code path for AESNI, CLMUL, and AVX instructions.

Proof size and composition. Our code can be broken down into top-level functional specifications, top-level interface specifications, and proof scripts. The top-level specifications are what must be understood in order to understand the results of our proofs.

```
rotate59_slice_add_thm <- prove_folding_theorem
    {{ \x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18
            x19 x20 x21 x22 x23 x24 x25 x26 x27 x28 x29 x30 x31 x32 x33 x34
            x35 x36 x37 x38 x39 x40 x41 x42 x43 x44 x45 x46 x47 x48 x49 x50 ->
        (slice_59_5_0 (x0 + x2 + x3 + x4 + x5 + x6 + x7 +x8 + x9 + x10 + x11
                      + x12 + x13 + x14 + x15 + x16 + x17 + x18 + x19 + x20
                      + x21 + x22 + x23 + x24 + x25 + x26 + x27 + x28 + x29
                      + x30 + x31 + x32 + x33 + x34 + x35 + x36 + x37 + x38
                      + x39 + x40 + x41 + x42 + x43 + x44 + x45 + x46 + x47
                      + x48 + x49 + x50))
          # (slice_0_59_5 (x0 + (64 * x1) + x2 + x3 + x4 + x5 + x6 + x7 +x8
                          + x9 + x10 + x11 + x12 + x13 + x14 + x15 + x16 + x17
                          + x18 + x19 + x20 + x21 + x22 + x23 + x24 + x25 + x26
                          + x27 + x28 + x29 + x30 + x31 + x32 + x33 + x34 + x35
                          + x36 + x37 + x38 + x39 + x40 + x41 + x42 + x43 + x44
                          + x45 + x46 + x47 + x48 + x49 + x50))
        == (x0 + (64 * x1) + x2 + x3 + x4 + x5 + x6 + x7 +x8 + x9 + x10 + x11
            + x12 + x13 + x14 + x15 + x16 + x17 + x18 + x19 + x20 + x21 + x22
            + x23 + x24 + x25 + x26 + x27 + x28 + x29 + x30 + x31 + x32 + x33
            + x34 + x35 + x36 + x37 + x38 + x39 + x40 + x41 + x42 + x43 + x44
            + x45 + x46 + x47 + x48 + x49 + x50) <<< 59 }};
```

Fig. 4. Example rewrite rule. This rule is made much more complex by the fact it happens after SAW's existing term optimization phases.

We have 168 lines of top-level interface specifications, which define the 8 interface functions that we've proved correct. Those functions specify memory layouts for the interface functions and link them to the top-level functional specification. We have 435 lines of top-level functional specifications, which were only slightly modified from specifications that we and others have used in previous cryptographic verification projects. These are almost completely free of implementation details, and live in a specifications only repository, separate from the code. If the functional specifications were made any shorter, they would likely also be less readable, so we believe they are close to optimal for their purpose.

The proof scripts consist of 1286 lines of intermediate function specifications, rewrite rules, tactics, and proof running logic. These intermediate functions are both proved and checked each time they're called. As a result, they do not need to be understood or trusted in order to believe the top level results.

Following continuous reasoning practice [19], our proofs are integrated into the CI process for AWS-LC. We do not expect this code to change, but we have adopted this practice as part of a larger AWS-LC assurance effort, including code which does change more often. Quickcheck versions of our proofs run as GitHub actions that take around 25 minutes[7]. The complete version runs on private systems in 30 minutes, but using more cores and memory. A significant part of our proof and tool development effort was dedicated to making sure

[7] https://github.com/awslabs/aws-lc-verification/actions

proofs could run within a time budget acceptable for CI (typically 1 hour). For example, this sometimes required introducing overrides to break the proof into smaller segments.

Achieving trust in the proof. SAW is designed to increase confidence in software, but it cannot supply total certainty. A key question is therefore what parts of the toolchain and proof must be trusted. For our proofs, the Trusted Code Base (TCB) consists of:

– The top-level functional and interface specifications in the proof scripts.
– Library behavior that is assumed and then used in overrides. In our case, that `OPENSSL_malloc` and `OPENSSL_free` behave correctly.
– The SAW and Cryptol toolchain, the tools themselves, the language models of x86 and LLVM, the back-end SMT solvers, and ultimately the Haskell runtime and other downstream infrastructure.
– Correctness of the compilation chain from LLVM to executable (for C code), and correct execution of compiled code by the hardware.
– Any behaviours of code not covered by proofs at the fixed sizes we have verified.

Although this TCB is significant, it is comparable in scale to similar verification projects like EverCrypt [21]. The highest impact improvement would likely proving the algorithms at arbitrary sizes. While we could throw computation at running the proofs at a wide range of fixed input sizes, this would spend computation and developer wait time with fairly little benefit. We believe an inductive approach is achievable in future work and would allow us to verify the algorithms once and for all.

In the mean time, we have covered some of the most-used block sizes, as well as all code paths. Given we have verified the algorithm at fixed sizes, and the code does not branch on input size, the only place that bugs could remain is the looping behaviour at other sizes. We have inspected the dynamically bounded loops in the code carefully to mitigate this possibility.

We could also shrink the TCB by applying foundational techniques such as used in the Verified Software Toolchain project [3]. This would remove much of the need to trust the tool itself. However, we believe doing this would make it infeasibly expensive to develop a tool as complex as SAW, at least given current foundational techniques.

Another important question is whether a correctness failure in the toolchain could result in a proof that does not establish the result we expect. We believe the probability of this is quite low. Our current best defense against this failure of TCB is thorough testing and code review. SAW itself is a well-tested tool that has been used for many projects. The language models have also been tested, and it is unlikely that a behavioural bug could cause an incorrect specification to be verified. For this to occur, several failures would need to occur at once.

As an aside, the intent of the people doing the proof, and the precise nature of any external review should be considered when answering this question. A tool bug is unlikely to result in a proof that falsely appears correct, *assuming that* *the proof effort is done in good faith.* On the other hand, all tools have bugs, and in most logic-based tools, bugs can allow the construction of false proofs that appears superficially correct. In other words, for most current tools, trust in verified code requires trust in the team and process producing the proof.

We believe the highest risk of accidental error lies in the specifications. It is quite common for draft specifications to contain subtle discrepancies between what users intends and the specification's formal meaning. We mitigate this with extensive manual audit. Every line of code we write is reviewed at least once within the verification team, and once by AWS-LC domain experts. The internal review ensures that our specifications are correct and that our style is consistent with our guidelines. The external review allows us to ensure that we have explained our proofs correctly, and that we have correctly specified the functions in the context that they are being used.

Proof engineering process. The proofs were completed over six calendar months, using approximately nine person-months engineering effort total. We consider this to be an upper bound estimate as the proof effort was mixed in with tool improvements, in particular for the less-mature x86 tooling. The core team consisted of four engineers, with additional contributions from verification tooling experts and AWS-LC domain experts. This project represented a significant engineering effort, but for our project, this represented a good use of resources to achieve a high level of confidence in the AWS-LC code. Proofs were completed alongside more traditional assurance approaches, e.g. testing, fuzzing, and code audits.

New proof techniques and tooling were a factor in our success, but there is no single technical breakthrough that made these proofs possible. While combined x86 and C verification is challenging, it would likely be possible (although not easy) to add such a capability to a number of existing tools. Rather, a series of tool extensions, design choices, and engineering working practices combined to make the project feasible.

Using SAW, we automated most of the trivial reasoning, which meant that a majority of proof engineering was spent on legitimately difficult verification problems. These mainly involved understanding the code being verified and using rewrites to manually rearrange verification terms to make them amenable to automated proving. Many of these steps could in principle be automated, but in practice engineers sometimes needed to resort to clunky debugging measures. We find it unsurprising that highly specialized code such as AWS-LC would generate edge cases that challenge generic proof automation. For proofs of this type, for now we believe completely automated proving is out of reach.

We take several steps to try to minimise engineer effort when building proofs. The most important of these is to lean on automation wherever possible. One example is that we try to avoid internal specifications, which are often the most challenging part of the proof. Because SAW is a bounded verifier, internal specifications are just a performance optimization—given sufficient compute resources, we could in principle symbolically execute the entire code-base. Of course, in practice, internal specifications are needed to make the proof tractable. Our practice is to prove functions at the largest scope which fits within our time budget. By doing this, we are sometimes able to avoid specifying internal functions that do relatively little computationally.

Another important strategy for us is to separate memory-safety proofs from functional correctness proofs. We have found that much of the technical risk in a verification project can be eliminated at the memory-safety stage. This is where the verification tools are most likely to run into show-stopping bugs that will put success of the project in jeopardy. Separating these concerns results in proof terms that are smaller and easier to understand, so bugs are easier to diagnose. Then, if we run into challenges during the correctness proving phase, we can limit the cause to correctness properties, eliminating a large fraction of the proof from consideration.

An important factor that enabled us to carry out these proofs is a team of expert proof engineers who have built their skills over years. This project was undertaken by a team which has worked continuously on verification projects for four years. This expertise has given us a better understanding of what we can attempt, and a far wider toolkit to dip into when things go wrong. We have seen, anecdotally, similar evidence of improved verification capabilities from other long-standing teams—for example, for the Project Everest, SeL4, and CompCert projects. Long-standing teams of proof experts are still unusual, but we believe they will be necessary to achieve the most ambitious proof engineering tasks, just as they are in software and tool development.

A significant lesson that we have learned about proof engineering is that a tool's behaviour when it fails is more important than success. This is a critical aspect of verification tools often overlooked in research papers. Many tools show a demo where everything works, but in a proof engineering effort, the vast majority of time is spent with a proof that does not work. In that sense, one of the most critical aspects of a verification tool is what it does when the proofs are not working. SAW provides some support for diagnosing errors, but there is a lot of room for improvement. It lacks tooling to allow proof engineers to easily inspect and modify proof terms that are not successfully proving. Furthermore, it has inefficiencies that can make repeatedly running and modifying proofs slow and painful, increasing the pain of developing proofs and reducing the time that engineers can spend on the real challenges of verification.

7.1 Trade-Offs When Building on Existing Verification Tools

As we saw in section 6, rewriting is an example where SAW's existing tooling made some parts of our proof more awkward. It is reasonable to wonder whether we could have modified SAW to allow more control of the rewriting pipeline. This highlights an interesting trade-off that exists when developing proofs using a more mature tool like SAW.

SAW has existed for a decade, and has been developed and improved iteratively over this time. Design decisions such as the order in which optimizations occur can sometimes be baked deeply into the tool. This stands in contrast to more experimental tools which often have short histories and a relatively clean design that can be torn down and refactored easily. SAW also has an active user community which relies on it for different verification and assurance tasks. The main users are at Galois, Amazon Web Services, and in the US government. This means that tool changes need wider approval from a community. Again, this stands in contrast to research tools which often have a single designer who is also the main user. The effect of this is that changes such as the introduction of rewriting must be carefully designed to fit with SAW's existing architecture.

The pay-off for these restrictions is an enormous increase in the power and scope of what we can achieve with the tool. In the large, we have benefited from many features that were developed by independent research teams. For example, we rely on the Macaw decompiler, which we used off-the-shelf without modification. The SAW LLVM semantics is likewise a product of many years of research, which did not require any further work from us. In the small, SAW embodies many, many clever tricks and pieces of good design that together make verification of challenging problems more feasible. Sometimes working with a mature tool imposes costs, but overall we believe it raises the bar for our work in a way that easily justifies the cost.

An open question for us is how we can make such collaboration possible across the verification community. Boogie [4] is a good example of a verification technology that has seen use across different teams and institutions. Proof assistants and SMT solvers are also widely used as a basis for new tools. However, there are still very few software verification tools that have seen significant adoption. We believe such tools will be necessary in the future if we collectively are to tackle larger and more complex verification problems.

7.2 Verified Code Generation versus Verifying Existing Code

While the approach in this paper results in an artifact that may appear externally similar to other state-of-the art verified cryptography efforts, there are some engineering factors that might influence which approach is most appropriate for a particular cryptographic use-case. The approach of EverCrypt, Jasmin, Fiat, and similar efforts require the user to produce code or a model in a language that is specific to the verification system. While these systems have demonstrated ability to produce efficient verified implementations, they cannot directly verify existing code. Our approach verifies existing code without modification, and there are several engineering benefits to this.

The most significant reason to verify existing code is that producing new code or modifying existing code introduces risk. Modifying optimized cryptographic code is particularly risky because it is complex, and because an error could have a devastating impact on the security of the system. A software project may be unwilling to accept the risk of modifying mature code, even if the new code is formally verified. For example, OpenSSL and its variants have been tested and audited over more than a decade, and this maturity is appealing to many software projects. In our approach, the code is verified without any modification. Zero new risk is introduced, and the verification process only increases trust in the system. A related benefit is that the verified code maintains any existing certifications, such as FIPS 140-2.

Another benefit of verifying existing code is that the verification works on the programming languages that are already used in the project, and the build pipeline does not require additional compilers or other tooling to support the language of the verification system. Having the build depend on this tooling can be risky because it is less familiar and less mature compared to the compilers and build systems that are typically utilized. There is a risk that a build pipeline could break or produce incorrect machine code due to a bug or lack of understanding of the verification system. In contrast, our approach produces a verification pipeline that is parallel to the build pipeline, and a failure in this pipeline does not have any impact on the main build pipeline.

Many cryptographic applications do not have any legacy concerns and never plan on maintaining or improving cryptographic code by hand. In those cases, EverCrypt, Jasmin, and Fiat all produce trustworthy, high-performance implementations that might prove easier to use and understand than what is provided by OpenSSL and its variants. Long-term support might be a concern, given these are research tools. However the slow-moving nature of cryptographic code makes it less likely that the implementations would need modification in the future.

8 Conclusion and Future Work

The purpose of formal verification is to allow users to be confident in the software on which they depend. This is the reason that AWS-LC, BoringSSL, and OpenSSL are excellent targets for formal verification. Nearly everyone who uses the internet relies on this code for security, either through end-user software, or through a cloud provider's infrastructure. Our proofs show for the first time that this kind of highly optimised, hand-written code matches its mathematical specification. More importantly, we show that such code can be verified for a reasonable amount of proof engineering effort.

We do not consider our proofs the last word on this code—there are several ways in which our work can be improved. Most importantly, we have not yet verified the OpenSSL version of AES-256-GCM and SHA-384. Based on inspection of the code, we believe the proofs would only need small changes to the term rewrites, but this is currently not a high priority in comparison to further AWS-LC assurance work.

There are also several ways we could improve the proofs themselves. We have verified this code at fixed input sizes. We believe we have covered all edge cases, so the probability that bugs remain is low, but a size-agnostic proof would be more complete. Our proofs also rely on term rewriting tactics to close the gap between implementation and specification. These rewrites are specialized to our application and are therefore the most fragile part of the proof. We believe that, with further research, automated solvers could solve many of these logical queries without the need for manual tactics (this would also make our proofs less fragile against code change). Finally, our proofs say nothing about non-functional security properties, such as timing or architectural side channels, nor do they connect to cryptographic security proofs.

We are at an exciting moment for cryptographic verification. It is now possible to deploy verified cryptography without compromising on performance. We are tantalisingly close to a world where most cryptographic traffic originates from verified code, and where new cryptographic primitives are verified as a matter of course. For our part, we consider AES-256-GCM and SHA-384 a stepping stone to the real prize: a fully verified library of production-grade cryptographic primitives. Stay tuned!

References

1. Almeida, J.B., Barbosa, M., Barthe, G., Grégoire, B., Koutsos, A., Laporte, V., Oliveira, T., Strub, P.: The last mile: High-assurance and high-speed cryptographic implementations. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 965–982. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00028
2. Amazon Web Services: AWS libcrypto (AWS-LC) public preview. https://github.com/awslabs/aws-lc
3. Appel, A.W.: Verified software toolchain. In: Goodloe, A., Person, S. (eds.) NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7226, p. 2. Springer (2012). https://doi.org/10.1007/978-3-642-28891-3_2
4. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005). https://doi.org/10.1007/11804192_17
5. Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K.R.M., Lorch, J.R., Parno, B., Rane, A., Setty, S., Thompson, L.: Vale: Verifying high-performance cryptographic assembly code. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 917–934. USENIX Association, Vancouver, BC (Aug 2017)
6. Brayton, R.K., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P.B. (eds.) Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6174, pp. 24–40. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_5
7. Chudnov, A., Collins, N., Cook, B., Dodds, J., Huffman, B., MacCárthaigh, C., Magill, S., Mertens, E., Mullen, E., Tasiran, S., Tomb, A., Westbrook, E.: Continuous formal verification of Amazon s2n. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification. pp. 430–446. Springer International Publishing, Cham (2018)
8. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)

9. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic – with proofs, without compromises. In: Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19) (May 2019)
10. Fromherz, A., Giannarakis, N., Hawblitzel, C., Parno, B., Rastogi, A., Swamy, N.: A verified, efficient embedding of a verifiable assembly language. Proc. ACM Program. Lang. **3**(POPL), 63:1–63:30 (2019). https://doi.org/10.1145/3290376
11. Galois Inc: Cryptol: The language of cryptography. https://cryptol.net/files/ProgrammingCryptol.pdf
12. Galois Inc: Macaw binary analysis framework. https://github.com/GaloisInc/macaw
13. Galois Inc: SAW tutorial. https://saw.galois.com/tutorial.html
14. Galois Inc: Software analysis workbench (SAW). https://saw.galois.com/
15. Galois Inc: What4 symbolic formula representation and solver interaction library. https://github.com/GaloisInc/what4
16. Google: boringssl. https://boringssl.googlesource.com/boringssl
17. Lim, J.P., Nagarakatte, S.: Automatic equivalence checking for assembly implementations of cryptography libraries. In: Kandemir, M.T., Jimborean, A., Moseley, T. (eds.) IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019. pp. 37–49. IEEE (2019). https://doi.org/10.1109/CGO.2019.8661180
18. National Security Agency: Commercial national security algorithm suite. https://apps.nsa.gov/iad/programs/iad-initiatives/cnsa-suite.cfm
19. O'Hearn, P.W.: Continuous reasoning: Scaling the impact of formal methods. In: Dawar, A., Grädel, E. (eds.) Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018. pp. 13–25. ACM (2018). https://doi.org/10.1145/3209108.3209109
20. OpenSSL cryptography and SSL/TLS toolkit. https://www.openssl.org
21. Protzenko, J., Parno, B., Fromherz, A., Hawblitzel, C., Polubelova, M., Bhargavan, K., Beurdouche, B., Choi, J., Delignat-Lavaud, A., Fournet, C., Kulatova, N., Ramananandro, T., Rastogi, A., Swamy, N., Wintersteiger, C.M., Béguelin, S.Z.: Evercrypt: A fast, verified, cross-platform cryptographic provider. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 983–1002. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00114
22. Ye, K.Q., Green, M., Sanguansin, N., Beringer, L., Petcher, A., Appel, A.W.: Verified correctness and security of mbedtls HMAC-DRBG. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 2007–2020. ACM (2017). https://doi.org/10.1145/3133956.3133974
23. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: A verified modern cryptographic library. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1789–1806. ACM (2017). https://doi.org/10.1145/3133956.3134043