Formal Verification of Cryptographic Software at AWS - Current Practices and Future Trends

Rod Chapman, Adam Petcher, Torben Hansen, Yan Peng,
Tancrède Lepoint, Cameron Bytheway, Panos Kampanakis
Amazon Web Services

Corresponding author: Rod Chapman (rodchap@amazon.com)

April 2024

Introduction

It appears that the winds-of-change are blowing for formal verification of cryptographic software and, furthermore, those winds are blowing in the same direction. We perceive progress on several fronts:

- Researchers in the mathematics of cryptography now publish formal specifications and proofs of security properties as a matter of course in their papers using language like EasyCrypt[10].
- Formal (yet executable) specification languages for cryptographic algorithms, such as Cryptol[11], are finally achieving acceptance and wider use within industry and government.
- There has been substantial progress in automated synthesis and verification of cryptographic software, including the work of Fiat Crypto[12], the Jasmin language and toolset[13], Hax[14], our own efforts, and many others.
- Governments are other standard-setting bodies are recognizing the importance of memory- and type-safe programming for critical applications.
- “Evidence-based” or “Principles-based” assurance[8] is gaining ground, following many years of use in the safety-critical domain.
- IETF have recently stood up a new “Usable Formal Methods Research Group”[15] to explore how formal notations and methods can improve IETF’s work in the future.

In light of these trends, the time is right to open discussion on how formal verification can influence the future development, verification and certification of cryptographic software.

Formal Verification of Cryptographic Software today at AWS

At AWS, we use Automated Reasoning (AR) to support the verification of a number of cryptographic services and libraries. This section presents a brief tour of each, showing a snapshot of how AR is used today:

AWS-LC

AWS LibCrypto (AWS-LC)[1] is an open-source cryptographic library, based on a fork of BoringSSL, with specific features and performance improvements

developed to support AWS needs, particularly on x86_64 and AArch_64 (e.g. AWS Graviton) processors. It is certified as a FIPS 140-3, level 1 cryptographic module. The library is written in a combination of C and assembly language. For the most commonly used and recommended algorithms, parameter sets and key lengths, we aim for verification of full functional correctness with respect to a formal specification using the following technologies:

- For establishing mathematical properties of the elliptic curves, we use the Coq proof assistant, working on the Cryptol specification.
- Proofs for C and x86_64 assembly language that was inherited from BoringSSL are developed using Galois' SAW toolchain, using specifications expressed in Cryptol.
- Verification of AArch_64 assembly language is performed using an in-house symbolic simulation tool, verification condition generator, and proof engines including multiple SMT solvers and Lean4. Again, we use Cryptol as the top-level specification in these cases.

For low-level elliptic curve and RSA functions, we use a new library called "s2n-bignum"[2] that is hand-written in assembly language for both x86_64 and AArch_64, and verified using the HOL-Light proof assistant. s2n-bignum is also designed to offer cryptographic-constant-time, and provides performance that matches or exceeds all other contemporary implementations for the micro-architectures that dominate AWS workloads. The library is permissively licensed, freely available, and the repository contains all the sources, HOL scripts, and continuous integration scripts to reproduce the proofs from scratch.

s2n-tls

s2n-tls is a clean-slate implementation of the TLS protocol stack. Its development and proof were carried out hand-in-hand, using Cryptol and SAW[3] for functional correctness, CBMC[18] for memory- and type-safety proofs, and SideTrail[19] for verification of constant-time properties. s2n-tls can use AWS-LC for its cryptographic primitives.

s2n-quic

Similarly, s2n-quic is a new implementation of the QUIC protocol, built atop s2n-tls and AWS-LC. It is written in Rust and deploys a hybrid verification style, where the Kani[4] verification tool is used to verify memory-safety, type-safety, and key correctness properties that lie beyond Rust's "borrow checker", combined with extensive dynamic verification using KATs, constrained fuzzing[5] and so on. We also use an automated traceability analysis tool called Duvet[17] to trace code to test cases and requirements to make sure that requirements from the QUIC specification are not overlooked.

Verified sampling for cryptographic computing

AWS Clean Rooms is a platform to collaboratively analyze datasets between multiple entities. To protect the contribution of any individual's data in aggregated

insights with differential privacy, it uses the SampCert sampler[12], a proven correct discrete Gaussian sampler implementation developed in Lean by AWS.

Options and ideas for the future

We anticipate and encourage further deployment of AR in cryptography. In particular, we see AR providing additional trust and assurance for our customers and regulators, plus the potential to improve the latency of FIPS evaluation (or any other evaluation scheme for that matter.). This section proposes a number of forward-looking ideas. In no particular order:

Verified memory- and type-safe programming

In line with guidance from US Government[6] and other national bodies, we anticipate a gradual shift to the use of programming approaches that facilitate sound formal verification of basic correctness properties such as memory-safety and type-safety. This style of development is well-established in the regulated safety-critical systems community (e.g. aerospace, nuclear, and rail), and some early experiments with cryptographic code has shown its feasibility for reference implementations[7].

Open and Evidence-Based Assurance

Again, this is a common practice in the safety-critical systems community. We can see a future where a submission for evaluation will include all design and verification artefacts including code, tests cases, proofs and the necessary tools to allow a recipient to reproduce the entire assurance case. A submission could take the form of a virtual machine image or a “Docker” file that contains all the artefacts and can be delivered electronically. We also anticipate that source code and proofs would be unchanged for multiple target platforms, allowing certification of a single module on multiple platforms to be achieved much faster than at present. This seems compatible with recent guidance from UK NCSC on “Principles-based Assurance”[8]. We hope that (re-)certification would be noticeably faster using this model, but only if regulators and test labs are willing to accept and give credit for AR-generated evidence. This brings us to...

Soundness cases for tools

In presenting such evidence, some sceptical person will always say “Ah... but why should I trust your proof tool?” Good point. We should advocate and encourage tool vendors to produce and make available their own assurance case. This requires a commercial and competitive market to emerge for verification tools, compilers and so on. Some verification tools and compilers have reached this level of maturity in the safety-critical sector.

Lightweight formal methods and hybrid verification

There still exists a perception that “formal methods” is an all-or-nothing activity, painful, and limited to a few specialist practitioners. We disagree. There is room for lighter styles of verification that can offer practical benefit without sacrificing soundness. Lightweight methods also tend to be faster, so can fit within a constructive “verify first” development style. Based on risk, there is also room for a

"hybrid" verification style that combines formal (static) verification of key correctness properties such as type-safety with dynamic verification (e.g. testing) for functional correctness using KATs, a reference Oracle (e.g. an executable specification in Cryptol), fuzzing and so on.

Equivalence proof and automatic optimization

Proof of equivalence of two (simple) programs is a well-established field in formal verification. Its utility in cryptographic software extends to proving that a (simple, easy-to-prove) reference implementation of a particular algorithm is functionally equivalent to a faster, more optimized version. Furthermore, we have encouraging results with the use of automatic optimization of assembly language[9], combined with "before and after" proofs of functional correctness. This removes some of the reliance on humans to hand-optimize the most time-critical functions. We can also foresee utility in proving the equivalence of a single module for multiple targets

Wider definition of "cryptographic module" to include novel applications

We are actively involved in many novel applications of cryptography, such as cryptographic computing, zero-knowledge proofs, fully homomorphic encryption, and differential privacy. Will the scope of FIPS certification expand in the future to cover these applications and their implementation in software?

Conclusion and Open questions

We hope this position paper inspires debate at the workshop and subsequent action. Open questions include:

- How will we (as a community) build trust with formal verification approaches and tool vendors?
- How will the results of automated reasoning tools be presented to customers, labs and regulators?
- How will the assumptions that underpin formal verification be presented and challenged?
- What incentive is there for test labs to accept and evaluate formal evidence?
- What are the training needs for NIST, the test labs and developers?
- Could the community run a "trial evaluation" of one or more formally verified cryptographic algorithms, giving the developers freedom to present and defend whatever formal evidence they choose? For example, there are at least 3 implementations of MLKEM that claim formal verification of various correctness properties.

References

[1] AWS-LC repository: https://github.com/aws/aws-lc
[2] s2n-bignum repository: https://github.com/awslabs/s2n-bignum
[3] 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 and Weissenbacher, G, (eds.) Proceedings of International Conference on Computer Aided Verification CAV 2018. (pp. pp. 430-446). Springer. DOI: 10.1007/978-3-319-96142-2_26.

[4] The Kani Rust Verifier. https://model-checking.github.io/kani/
[5] Bolero - a fuzz and property testing front-end for Rust. https://github.com/camshaft/bolero
[6] Back to the Building Blocks: A Path Toward Secure and Measurable Software. US White House. February 2024. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf
[7] AWS LibMLKEM repository. https://github.com/awslabs/LibMLKEM/
[8] Principles-Based Assurance site. https://www.ncsc.gov.uk/information/principles-based-assurance
[9] SLOTHY: Assembly optimization via constraint solving. https://github.com/slothy-optimizer/slothy
[10] The EasyCrypt language and toolset. https://formosa-crypto.org/projects/easycrypt
[11] Cryptol - The Language of Cryptography. https://cryptol.net/
[12] SampCert: verified implementation of discrete Gaussian sampler https://github.com/leanprover/SampCert
[13] Fiat-Crypto: Synthesizing Correct-by-Construction Code for Cryptographic Primitives. https://github.com/mit-plv/fiat-crypto
[14] Jasmin language and toolset. https://formosa-crypto.org/projects/jasmin
[15] Hax - A Rust Verification Tool. https://hacspec.org/
[16] IETF Usable Formal Methods Research Group. https://datatracker.ietf.org/rg/ufmrg/about/
[17] Duvet repository: https://github.com/awslabs/duvet
[18] CBMC - The C Bounded Model Checker. https://github.com/diffblue/cbmc
[19] Athanasiou K, Cook B, Emmi M, MacCarthaigh C, Schwartz-Narbonne D, Tasiran S. SideTrail: verifying time-balancing of cryptosystems. In: Piskac R, Rümmer P, eds. Verified Software. Theories, Tools, and Experiments. Cham, Switzerland: Springer International Publishing; 2018:215-228 https://doi.org/10.1007/978-3-030-03592-1_12.