A Formally Verified Cryptographic Extension to a RISC-V Processor

Joseph R. Kiniry
kiniry@galois.com
Galois, Inc.
Portland, OR, USA

Daniel M. Zimmerman
dmz@galois.com
Galois, Inc.
Portland, OR, USA

Robert Dockins
rdockins@galois.com
Galois, Inc.
Portland, OR, USA

Rishiyur Nikhil
nikhil@bluespec.com
Bluespec, Inc.
Framingham, MA, USA

ABSTRACT

Security in computing systems is based upon having correct and secure software, firmware, and hardware. Formal reasoning is capable of providing solid assurance about correctness and security for software and firmware, and formal reasoning about hardware correctness has advanced significantly in recent years. However, formal reasoning about *system security* is in its infancy. The security of software systems is often reliant upon cryptographic foundations and development artifacts open to peer review, such as formal protocol and algorithm specifications and proofs of correctness. The security of hardware systems, however, is virtually always based upon secrecy and limited amounts of testing. RISC-V presents an opportunity to change the state of system security assurance for the better. As a first step in this direction, we describe a formally verified cryptographic extension to RISC-V and its assurance case.

KEYWORDS

RISC-V, cryptography, assurance, verification, applied formal methods

ACM Reference Format:
Joseph R. Kiniry, Daniel M. Zimmerman, Robert Dockins, and Rishiyur Nikhil. 2018. A Formally Verified Cryptographic Extension to a RISC-V Processor. In *Proceedings of Second Workshop on Computer Architecture Research with RISC-V (CARRV 2018).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

For a computing system to be secure, it must have correct and secure software, firmware, and hardware. Formal reasoning and rigorous development techniques are capable of providing solid assurance about software and firmware correctness and security. Formal reasoning capabilities for hardware correctness have advanced significantly in recent years, and are becoming more widely used. However, formal reasoning about hardware security and system security (i.e., the security of particular combinations of software, firmware, and hardware) is in its infancy.

In software, security assurance cases typically rely upon well studied cryptographic foundations and development artifacts open to peer review, such as formal protocol specifications, formal algorithm specifications, and proofs of correctness. In hardware, however, security assurance is nearly always based upon secrecy and limited amounts of testing, with no formal proofs and minimal, if any, peer review.

The open RISC-V ecosystem presents an opportunity to change the state of system security assurance for the better, by enabling for hardware the open peer review and independent formal verification already available for high assurance software. As a proof of this concept we have built a formally verified cryptographic extension to RISC-V, incorporated it into a small system, and developed an assurance case spanning the system's hardware, firmware, and software. Here, we primarily describe the cryptographic extension and the assurance techniques we used to verify its correctness, as compared to typical assurance techniques used for validated cryptographic systems.

2 CRYPTOGRAPHIC IMPLEMENTATION ASSURANCE

2.1 The Status Quo

Assurance in today's hardware and software cryptographic implementations is primarily achieved through validation against national standards like the U.S. Federal Information Processing Standard 140-2 [11] (FIPS 140) and associated Implementation Guidance [12]. FIPS 140, which has not been updated since 2001, has 4 levels (called simply "Level 1" through "Level 4"). At the lowest security level (Level 1), validation requires the presence of structured documentation of various kinds for the system's cryptographic functionality, and that the system passes all relevant tests mandated by the NIST Cryptographic Algorithm Verification Program (CAVP). At higher security levels, validation also

CARRV 2018, June 2, 2018, Los Angeles, California, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

requires physical tamper-evidence and role-based authentication (Level 2); physical tamper-resistance, identity-based authentication, and a separation between the module's interfaces for "critical security parameters" and its other interfaces (Level 3); and stricter physical security requirements including robustness in the face of environmental attacks (Level 4). At no security level does FIPS require any resistance to side-channel attacks (e.g., differential power analysis).

CAVP tests—the only part of the FIPS standard that touches upon the *correctness* of cryptographic algorithm implementations—consist of large sets of test vectors, one for each cryptographic algorithm under test, generated on-demand by an independent test laboratory ("lab") for each evaluation. The lab supplies these test vectors to the vendor of the cryptographic module, who runs the module against those vectors and returns the results to the lab. The lab then evaluates the results and validates each cryptographic algorithm that performs correctly on all test vectors.

Unfortunately, CAVP testing covers only a minuscule fraction of the state space. A typical set of vectors for the AES-128 block cipher contains anywhere from 20 to 1000 key and block pairs. With a key size of 128 bits and a block size of 128 bits, this means that the largest sets of test vectors cover on the order of $10^{-72}\%$ of the state space. CAVP testing also relies entirely on the vendor to run test vectors through the actual implementation being validated, as opposed to submitting results generated by another, already known correct implementation to "game" the validation process.

In practice, many cryptographic implementations have received FIPS validation and been later found to be insecure. For example, OpenSSL has had FIPS validated versions since 2012, but many of these were vulnerable to the Heartbleed bug [13]. Moreover, once a cryptographic module is FIPS validated, *any change*—no matter how small or critical—requires revalidation. In practice, this means that when a bug like Heartbleed is discovered, FIPS validated cryptographic modules either *remain vulnerable* for months until they are revalidated with new mitigations, or *lose their validation* when patched. Of course, FIPS validated hardware modules that cannot be patched remain vulnerable for as long as they are deployed.

2.2 Applied Formal Methods for Cryptography

Using today's applied formal methods tools, we can formally verify both correctness and security properties of cryptographic models, in the form of algorithms and protocols, and their implementations in both hardware and software.

Models: Algorithms and Protocols. Algorithms and protocols can be formally specified and mechanized using several different formal methods tools. The appropriate choice of tool depends upon several factors: (1) what kind of reasoning one wants to do about the model; (2) whether or not one wants to relate the model to concrete implementations; (3) what hardware design languages or programming languages those implementations are written in; and (4) whether or not one wants the specification to be executable.

Verification of correctness or security properties is either symbolic—reasoning about the model from a logical point of view—or computational—reasoning about adversarial capability. For the most important algorithms or protocols, one typically wants to perform both kinds of reasoning.

Models are used as reference specifications for the formal verification or rigorous validation of hardware or software implementations. In addition to reasoning about models, one can use them in a variety of other (typically automated) ways, including synthesizing implementations, generating testbenches, measuring coverage, performing runtime verification via bisimulation, and more. Implementations are realized either in software or hardware, as discussed in the next two subsections.

Software. Much like reasoning about models, one can reason about software implementations in a variety of different ways. Is an implementation correct with respect to a given model? Does an implementation have specific kinds of security vulnerabilities, such as a timing side-channel?

Several different formal methods techniques, each of which has different sub-flavors, are used to reason about implementations; these include abstract interpretation, logic-based reasoning, model checking, and runtime verification.

Academic and commercial tools have good support for reasoning about a few very popular programming languages and platforms, such as C, Java (both the programming language and virtual machine), and Microsoft's C#, Spec#, and F# programming languages and .NET platform. A number of assurance-focused programming languages and platforms, such as SPARK, also have good reasoning support.

Our recent public case studies in these matters include work on synthesizing formally verifiable high-performance cryptographic implementations in C or LLVM [8] and work on formally verifying Amazon's s2n TLS library [6].

Hardware. The state of the art in formal verification of cryptographic implementations in hardware is considerably less pervasive in academic literature, and profoundly less transparent.

The vast majority of companies that implement hardware cryptography provide no information whatsoever about the hardware's security or assurance. This is true of broadly deployed technology, including cryptographic instruction extensions to mainstream processors like Intel's AES-NI, enclaves like Intel's SGX, ARM's TrustZone, and Apple's Secure Enclave, and hardware roots of trust like the various implementations of the Trusted Platform Module standards. While this information is commonly available for virtually all mainstream software cryptography, it is almost entirely absent for specialized cryptographic hardware implementations such as co-processors, daughter cards, and hardware security modules from companies like Intel, Texas Instruments, Thales, and Utimaco.

Our experiments with open source and commercial tools have shown us that hardware "formal" tools are unlikely

to be able to reason about the correctness of even simple cryptographic algorithms. These tool limitations are not a fundamental problem with respect to certification and validation of cryptographic modules for federal government use under current standards (though they do, as we have noted, lead to the validation of modules that have egregious security flaws). However, these limitations *are* a significant problem for high-assurance secure systems, particularly those deployed by the Department of Defense and the Intelligence Community, which regularly hand-design cryptographic hardware implementations at the mask level precisely because they do not trust the capabilities, provenance, and assurance provided by industry's EDA tools.

2.3 The Near Future

The introduction of applied formal methods into the security validation process would represent a dramatic improvement over the status quo. Formal verification can provide much better assurance of algorithm correctness than small sets of test vectors, and can also detect information flow issues that lead to security vulnerabilities. Reusing assurance artifacts like proofs and automatically generated security testbenches also means that revalidation can have higher assurance with only differential cost and effort [7].

In order to have an impact on cryptographic hardware assurance, we must advance the state-of-the-art in formal verification tools for hardware verification. We tackle this problem by (1) shifting our perspective, (2) choosing to use automated synthesis over manual work whenever possible, (3) concurrently using a variety of tools and techniques to design, implement, validate, and verify cryptographic modules, and (4) bringing the best R&D from applied formal methods from the software world to hardware.

Our perspective on this challenge is that modern hardware engineering is very much like outdated software engineering. Thus, we can repurpose and adapt many recent research concepts and results in applied formal methods for software to the hardware space. The work summarized in this paper, that of formally verifying a cryptographic extension to a RISC-V processor, is exactly of this nature.

3 A FORMALLY VERIFIED AES EXTENSION TO RISC-V

Our proof of concept for the techniques discussed above is a formally verified AES extension to an existing RISC-V processor, Bluespec's Piccolo [1], which is implemented in Bluespec SystemVerilog (BSV). The RISC-V ecosystem is an ideal place for this sort of experimentation, for several reasons: the open source nature of the ISA itself; the wide variety of freely available IP; and rapidly growing interest in RISC-V on the part of both government and industry.

We extended Galois's existing formal methods tools with the ability to formally reason about properties of a core subset of the BSV language, and used the resulting tools to formally verify our AES extension.

3.1 Our AES Specifications and Implementations

We used multiple independent formal specifications of AES, and developed and verified multiple firmware and software implementations, in the course of this work. Our AES specifications are based on the NIST Advanced Encryption Standard [10] and are written in Cryptol [9], Galois's cryptography specification language. They describe two different "flavors" of AES—known as *S-box* and *T-box* architectures—that are semantically equivalent but have substantially different internal construction.

For software and firmware, we used open source off-the-shelf C implementations of AES as well as an implementation synthesized from one of our Cryptol specifications. Our hardware implementation was written by hand in BSV, mimicking the S-box-based Cryptol specification. We did not implement our AES RISC-V extension as a proper instruction set extension, due primarily to time constraints in the project; instead, we implemented it as a RISC-V co-processor that uses DMA to read/write key and text blocks from/to main memory, controlled and synchronized via memory-mapped CSRs.

3.2 Our AES Verifications

Galois's verification toolkit, the Software Analysis Workbench (SAW) [3], uses symbolic execution to translate implementations and specifications into formal models. In so doing, it executes code on symbolic inputs, effectively unrolling loops and translating the code into a circuit representation. Symbolic execution is well suited to verifying cryptographic algorithms, in part due to their frequent use of bounded loops. The most common use of SAW is to prove equivalence between Cryptol specifications of algorithms and concrete implementations of those algorithms.

In this project, we used SAW both to verify the mutual equivalence of all our Cryptol AES specifications and to verify that our BSV AES implementation correctly implements the Cryptol specifications. Because we had not previously performed any verifications of BSV code, we first needed to mechanize a core subset of BSV and incorporate that mechanization in an extension to SAW.

3.3 Rigorously Validating Synthesized Verilog RTL

In the space of applied formal methods, rigorously validating an implementation amounts to testing that implementation against a model in a fashion that finds the greatest number of bugs with the least human effort and computational cost. We generally perform such validation using three main techniques.

First, we can synthesize a testbench based upon the model and its implementations. Using a whitebox analysis framework such as PEX [15], one can automatically synthesize the smallest set of non-isomorphic tests that has the largest implementation coverage. Using a model-based testbench generator, such as JMLUnitNG [17], one can generate a testbench that respects the shape and nature of the model,

automatically identifying interesting modules, types, and values to cover all corner cases of the model.

Second, we can "fuzz" the implementation and its testbench using American Fuzzy Lop [16] and testing frameworks such as CUTE [5], KLEE [2], PITest [14], and QuickCheck [4]. These tools and techniques "level up" runtime verification testing tremendously, discovering design flaws, incorrect assumptions in models, and implementation bugs.

Finally, we can write a rigorously designed testbench to perform positive and negative property runtime verification. We took this approach for this case study, building a rigorously designed testbench whose scenarios are entirely driven by the abstract state machines of the software, firmware, and hardware layers. Given our experience in writing these kinds of testbenches, we can guarantee 100% architectural coverage, abstract state machine value and path coverage, scenario coverage, and function coverage.

3.4 Rigorously Validating RTL on an FPGA or ASIC

As discussed earlier, this case study uses a rigorously designed testbench to perform positive and negative property runtime verification of the system.

The exact same testbench can be used in five test scenarios: (1) to run the demonstrator at the model level in Linux; (2) to simulate the Bluespec design in the Bluespec event-based simulator; (3) to simulate the RTL in a Verilog simulator; (4) to execute the RTL on an FPGA, and (5) to execute the testbench on a fabricated ASIC on a test board. In this project, we did only (1) through (4), and did not fabricate an ASIC or a test board.

This layering of runtime verification testing means that we are performing multi-level bisimulation testing, answering the question "Does every execution at each runtime verification level perform identically?".

For the purposes of our AES extension to RISC-V, this framework is more than sufficient to provide high assurance of the implementation's correctness. The reason we need not also use fuzzing or automatically synthesized testbenches, as discussed in the previous section, is that we are using two powerful complementary techniques to provide assurance. First, we are automatically synthesizing our implementation using the Bluespec BSV Compiler. Second, and more importantly, we are formally reasoning about the correctness of the BSV implementation and multiple firmware C and LLVM implementations against independent Cryptol specifications, as discussed in section 3.2. In addition, there is good reason to believe our implementation is correct given the nature of AES's design as a symmetric cipher (a topic we will not expand upon here).

4 CONCLUSION

To the best of our knowledge, the work we have described here represents the first public formally verified AES hardware implementation. The novelty of this work is that the formal verification is wholly automated, is very fast (the verification takes only a few seconds), and the AES that is verified is the entire AES-128 algorithm rather than just the innermost *round* function/feature exposed by Intel's AES-NI.

There are many opportunities for extensions to this work. First, we intend to extend our synthesis capabilities to support direct synthesis from Cryptol to hardware description languages (HDLs, e.g., Verilog, VHDL, and BSV) in addition to languages such as C and LLVM. Second, we intend to significantly extend our security verification capability's breadth (across product lines and HDLs) and depth (between abstraction layers, from abstract models down to RTL). We are also in the process of integrating multiple automatic testbench generation and verification techniques.

One aspect of assurance that we did not explore in this work, primarily because of time constraints, is verification that the synthesizable Verilog generated by the BSV compiler is correct with respect to the source BSV (and, transitively, with respect to its Cryptol specification). A verified BSV compiler could provide a formal link between Cryptol and the synthesizable Verilog and, in combination with existing hardware formal verification tools, could extend the assurance case down to the gate level..

Our hope is that the foundations we have described here will have an impact on both the standardization work of the RISC-V Foundation and the correctness and security of commercial RISC-V implementations. In particular, with regard to the Foundation, we are contributing to the formal specification and security technical committees and intend to contribute in a similar fashion to the vector extension, memory model, and compliance technical committees. For industrial applications, we look forward to companies applying these formal verification capabilities to their unique challenges.

ACKNOWLEDGMENTS

This research was sponsored by the Air Force Research Laboratory (AFRL) and developed with funding from the Defense Research Projects Agency (DARPA) under contract number FA8650-16-C-7665. Any views, opinions, findings, conclusions and/or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force, the Department of Defense, or the U.S. Government.

REFERENCES

[1] Bluespec. 2017. Piccolo RISC-V Core. http://bluespec.com/piccolo-risc-v-core/. (2017). http://bluespec.com/piccolo-risc-v-core/

[2] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of high-coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[3] Kyle Carter, Adam Foltzer, Joe Hendrix, Brian Huffman, and Aaron Tomb. 2013. SAW: the Software Analysis Workbench. In *ACM SIGAda Ada Letters*, Vol. 33. ACM, 15–18. http://dl.acm.org/citation.cfm?id=2527277

[4] Koen Claessen and John Hughes. 2011. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM Sigplan Notices* 46, 4 (2011), 53–64.

[5] CUTE - C++ Unit Testing Easier [n. d.]. CUTE - C++ Unit Testing Easier. http://cute-test.com/. ([n. d.]). http://cute-test.com/

[6] Joey Dodds. 2016. Verifying s2n HMAC with SAW. https://galois.com/blog/2016/09/verifying-s2n-hmac-with-saw/. (Sept. 2016). https://galois.com/blog/2016/09/verifying-s2n-hmac-with-saw/

[7] Radu Grigore and Michał Moskal. 2007. Edit and verify. *arXiv:0708.0713 [cs]* (Aug. 2007). http://arxiv.org/abs/0708.0713 arXiv: 0708.0713.

[8] Joseph R. Kiniry, Peter Beerel, William Koven, and Daniel M. Zimmerman. 2016. Galois Ultra Low Power High Assurance Asynchronous Crypto. In *Lightweight Cryptography Workshop 2016*. NIST, Gaithersburg, Maryland. https://www.nist.gov/sites/default/files/documents/2016/10/19/kiniry-presentation-lwc2016.pdf

[9] Jeffrey R. Lewis and Brad Martin. 2003. Cryptol: High assurance, retargetable crypto development and validation. In *Military Communications Conference. MILCOM*, Vol. 2. IEEE, 820–825.

[10] National Institute of Standards and Technology. 2001. *Announcing the AES Encryption Standard (AES)*. Technical Report NIST FIPS 197. National Institute of Standards and Technology, Gaithersburg, MD. https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf DOI: 10.6028/NIST.FIPS.197.

[11] National Institute of Standards and Technology. 2001. *Security requirements for cryptographic modules*. Technical Report NIST FIPS 140-2. National Institute of Standards and Technology, Gaithersburg, MD. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-2.pdf DOI: 10.6028/NIST.FIPS.140-2.

[12] National Institute of Standards and Technology. 2018. Implementation Guidance for FIPS 140-2 and the Cryptographic Module Validation Program. https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Module-Validation-Program/documents/fips140-2/FIPS1402IG.pdf. (Jan. 2018). https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Module-Validation-Program/documents/fips140-2/FIPS1402IG.pdf

[13] OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160) [n. d.]. OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160). https://www.us-cert.gov/ncas/alerts/TA14-098A. ([n. d.]). https://www.us-cert.gov/ncas/alerts/TA14-098A

[14] PIT Mutation Testing [n. d.]. PIT Mutation Testing. http://pitest.org/. ([n. d.]). http://pitest.org/

[15] Nikolai Tillmann and Peli de Halleux. 2008. Pex - White Box Test Generation for .NET. In *Tests and Proofs (TAP '08)*. https://www.microsoft.com/en-us/research/publication/pex-white-box-test-generation-for-net/

[16] Michal Zalewski. 2015. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. (2015). http://lcamtuf.coredump.cx/afl/

[17] Daniel M. Zimmerman and Rinkesh Nagmoti. 2010. JMLUnit: The Next Generation. In *Formal Verification of Object-Oriented Software (Lecture Notes in Computer Science)*, Vol. 6528. Springer, Berlin, Heidelberg, 183–197. https://doi.org/10.1007/978-3-642-18070-5_13