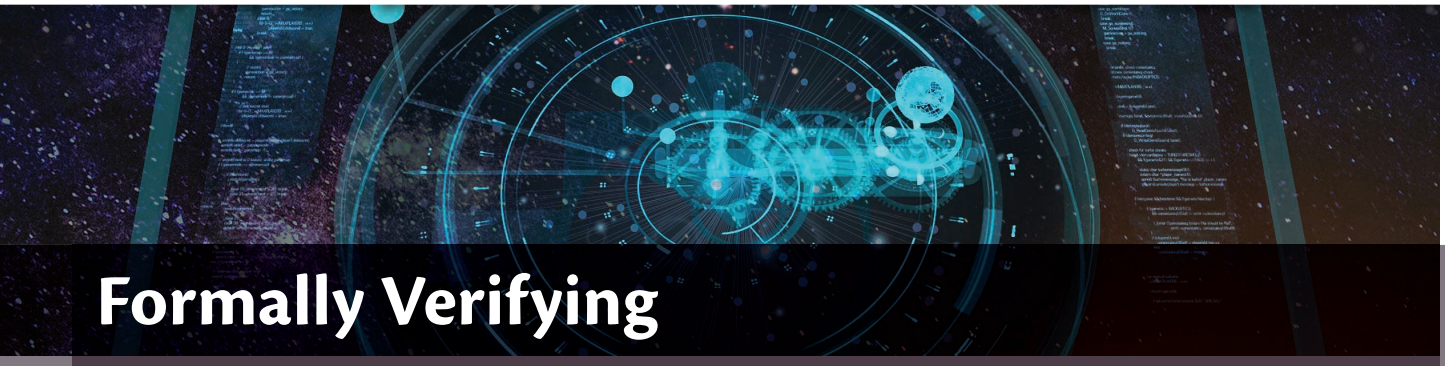

# Formally Verifying Industry Cryptography

**Mike Dodds** | Galois, Inc.

**Over the past five years, Galois has formally verified several cryptographic systems that are used in demanding industry environments. This article discusses our approach to these verification projects, focusing on the practical engineering challenges that exist when building and deploying proofs in industry.**

For the past five years, Galois has been formally verifying cryptographic software for use in demanding industry environments. Using tools developed over two decades, we have formally verified key properties of the s2n Transport Layer Security (TLS) stack,[1] the AWS LibCrypto library,[2] and the blst library.[3] These proofs are among the most complex ever deployed in industry, and the software they verify protects data confidentiality and integrity for hundreds of millions of users.

These proofs represent a significant technical and engineering accomplishment. The systems we target were not designed with verification in mind and could not be substantively modified. These systems are not static, and they are embedded into broader systems, which themselves are subject to change. The proofs were deployed into the workflows of highly dynamic engineering teams. We have developed tools and practices to overcome these challenges based on our verification tool-suite, the Software Analysis Workbench (SAW).[4]

This article discusses Galois's approach to cryptographic verification projects, focusing on the practical engineering challenges that exist when building and deploying proofs in industry. Readers curious about technical matters are directed toward our 2018 and 2021 research articles.[1,2] This article expresses Galois's approach to verification and does not represent the practices or opinions of any of our clients.

## A Tool in the Toolbox

We see formal verification as one tool for software assurance, alongside other methods such as testing, code audits, and fuzzing. Like many assurance tools, formal verification is best applied selectively to the pieces of a system where it will have the highest impact. For example, for the s2n TLS stack, we began by verifying the Hash-based Message Authentication Code (HMAC) and Deterministic Random Bit Generator (DRBG) cryptographic primitives and then moved to the TLS protocol code. We find that formal verification works best when applied to self-contained components that are particularly vulnerable or security-critical, such as parsers, protocol engines, core operating system functionality, and cryptographic systems.

Like other assurance tools, formal verification provides assurance evidence. The result of the verification process is a mathematical proof that can be automatically checked, which gives very high confidence that the code

*Digital Object Identifier 10.1109/MSEC.2022.3153035*
*Date of current version: 11 March 2022*

satisfies the desired properties in every specified situation. C, LLVM, or x86 machine codes are often amenable to practical formal verification, especially for two software properties of interest in commercial systems:

- The first is *safety*, meaning the absence of crashes, memory safety errors, race conditions, and so forth. In our projects, we most often prove that *undefined behavior* in the C and LLVM sense cannot occur.
- The second is *functional correctness*, meaning equivalence between the code and some reference specification. In our projects, a specification most often describes the cryptographic algorithm implemented.

Before embarking on a verification project, we encourage careful thought about the types of bugs or threats that verification will guard against. In our cryptographic verification work, safety and functional correctness address two different security concerns. Safety violations in a cryptographic module might allow an attacker to write to memory and thereby execute an exploit on the host device. Meanwhile, a lack of functional correctness might mean that the encryption has been performed incorrectly, potentially allowing attacks against message or protocol security.

Like other assurance tools, formal verification comes with a cost. Constructing such proofs requires a significant proof engineering effort. For a highly optimized system of the type that we typically consider, months of effort are usually required for thousands of lines of code. For example, verifying the AES-256-GCM and SHA-384 algorithms in AWS LibCrypto took approximately nine person-months of work by experienced Galois proof engineers.

Cost is a significant consideration when deciding where formal verification should be applied relative to other assurance techniques. We suggest that it should be targeted at the most critical threats to a system. The relative likelihood and impact of different threats cannot be ascertained completely formally. Rather, they must be determined in the context of the overall system and the engineering team developing it.

## Proofs of What?

Proofs are formal artifacts with exact mathematical meanings. However, we have discovered that the raw technical theorems are often of little value in explaining the outcomes of a proof. It is often necessary to contextualize proofs in terms that make sense to the software design team. Teams may have different goals for proofs: for example, increasing confidence in a system may require a different focus from finding bugs. We find that a large proportion of our interaction with engineering teams involves iterating on the precise properties that will be delivered.

All proofs have limitations to the guarantees they provide. One common type of limitation is that, even if a piece of code is verified, its dependencies and calling context generally are not. Verification may also assume that the compiler is correct or that microarchitectural or supply chain attacks are impossible. Proofs may also have technical limitations, such as assuming fixed data sizes or fixed numbers of loop iterations (this is true for some of Galois's proofs). As a result, *systems that have been formally verified may still contain bugs*. Proofs can increase confidence but cannot guarantee absolute certainty.

We take particular care to explain proof limitations in a way that can be understood by the teams we work with. If expectations are not set appropriately, a bug in a formally verified system can severely reduce trust in formal verification as a whole. This is true even if the bug appears outside the verified components of the system. One remedy for this, in our experience, is ensuring that proof limitations are explained in detail, with a minimum of jargon. We work with our clients to determine which proof limitations represent tolerable risks and which must be eliminated.

The riskiest elements of a proof are external specifications because these are the boundaries between the verified and unverified portions of the system. Our proofs require two types of specification: those for the system's dependencies and a high-level specification for how the system will be called by the context. *Dependency specifications* are assumptions about how libraries and other dependencies behave. If these are wrong, then (for example) a library call may fault or return an unexpected value. In contrast, the high-level specification is guaranteed by the proof but only so long as the system is called as expected. For example, a high-level specification might assume that a particular input is positive. However, if a negative value is passed by the calling context, then the proof makes no guarantees about the system behavior. In these situations, bugs may occur.

To avoid specification bugs, we take care to audit and (where possible) test specifications against real system behavior. We recommend choosing a specification style that can be audited by the teams consuming the proofs. Our specifications are written in a high-level, domain-specific language called *Cryptol*.[5] They are designed to be auditable by engineering teams with some background in mathematics and cryptographic algorithms. Where possible, we also target interfaces with stable and unambiguous specifications: for example, for AWS LibCrypto, we target the EVP interface, which is shared with OpenSSL and is intended as a common public interface for cryptographic primitives. This reduces the probability of bugs and misunderstandings.

## Verifying Preexisting Systems

Recent years have seen several high-assurance systems explicitly developed to be formally verified. For example, a C compiler (CompCert[6]), an operating system (SeL4[7]), and an HTTP stack (Project Everest[8]). These projects provide high-quality and highly reliable drop-in alternatives to unverified code, and we recommend them for many use cases. But in practice, most systems do not use built-for-verification code. To make our approach broadly applicable, we target preexisting systems with no (or only superficial) modifications. This has the effect of making proofs more challenging: our experience is that preexisting systems are dramatically more difficult to verify than built-for-verification systems.

The first source of difficulty in preexisting systems is the heavy use of optimization. Cryptographic systems contain some of the most heavily optimized code in existence. The reason for this is performance. A single cryptographic primitive running on a cloud platform may be called millions or billions of times per second. At scale, even tiny optimizations translate to big gains. We hypothesize that this will be true for many critical components of production systems. Once a system is deployed at scale, gains in performance will become increasingly valuable, creating pressure for heavier optimization.

Unfortunately, each optimization makes verification more complex because it represents a step away from the simple, obviously correct way to implement a system. Speaking loosely, each optimization step requires a small theorem showing that both the original and the optimized code are equivalent. Real cryptographic systems layer dozens or hundreds of optimizations, developed over many years. Optimizations may be valid only as a result of deep mathematical or probabilistic properties, and they may alter the natural interfaces of functions, complicating the task of specification. To verify the system, each small theorem must be proved. For many steps, this can be accomplished in an automated manner. But in our experience, many optimization steps require manual proofs.

The second source of difficulty in preexisting systems might be called *problem diversity*. Any verification project presents a series of proof tasks of different kinds: for example, supporting a particular library, modeling a particular CPU operation, defining a particular specification, or completing a particular reasoning step. Every verification tool has some proof tasks to which it is well suited. In our experience, the cost of verification is overwhelmingly caused by a small number of difficult proof steps. Preexisting systems inherently present a more diverse set of problems. This increases the frequency of the most problematic proof steps, resulting in more expensive proofs and more unpredictable costs. In some cases, the most problematic steps can be skipped. But this depends on which limitations on the proof are considered acceptable.

Another effect of problem diversity is that it is very difficult to predict up-front what tool capabilities will be necessary for a particular type of proof. As a result, it is risky to design a proof tool without access to real-world proof tasks. We believe that effective tools are built incrementally by actually undergoing proof projects. Almost every SAW verification project has required us to extend our tools. For example, verifying postquantum cryptography pushed us to improve the performance of our tools, and verification of TLS pushed us to develop new approaches for specifying and verifying state machines. SAW is built on a common symbolic execution engine called *Crucible*, which can be used for multiple languages and tools. This allows improvements to pay forward in subsequent projects.

Of course, this approach works only for teams that develop proofs and proof tools together. This is a common model for proof engineering as it is practiced in 2022—however, it is inherently nonscalable. If formal verification is to be used much more widely, we must reach a point where proof tools do not need to be updated for each new proof. We hope to mitigate this problem by improving our tools step by step. As we complete more projects, our tools increase in power, but also, more proof tasks become routine, requiring no tool modifications.

## Continuous Reasoning

Proofs are tools for assuring systems, and this means they must fit into the workflows of the engineering team building the system. Change is a constant in real engineering environments, and proofs, like tests, need to be repeated every time a design changes. It is essential to build proofs that handle change gracefully with minimal disruption to existing engineering practices.

One tactic that has proved successful is to build our proofs into continuous integration (CI), alongside the existing test suite. In doing this, we follow Peter O'Hearn and the Infer team at Facebook, who call this approach *continuous reasoning*.[9] Our proofs run on every commit to the codebase, and errors are raised at code review time. In one deployment, SAW has been running for more than four years over hundreds of code changes.

Running in CI has many advantages: O'Hearn[9] explains these well in his 2018 paper, so we will only review them briefly. The key advantage is that potential failures are flagged before they reach production. This is often the main benefit of formal verification for engineering teams: fewer mistakes and more guardrails. Because teams are under time pressure, the aim is often

to reduce assurance time and release software more rapidly. Proofs can grant confidence that a system can be modified and released without additional quality assurance processes.

Running in CI generates constraints that are often just as challenging as developing the proof itself. First, the proof tools must integrate into the build system and the CI system. These are remarkably diverse and may change more often than the code itself. A proof tool must run as a container inside the system and must produce results in a format that suits the error-monitoring stack as well as the team's preferred working practices. For SAW, we typically build and maintain custom support for CI integrations for each proof deployment.

Second, the proof must fit within CI time budgets: generally under an hour but often under 20 minutes. Large SAW proofs may take several hours to run, so we often have to decompose them into smaller proof units that can run in parallel. In the best case, we can solve this problem by adding function interfaces and verifying compositionally. But several engagements have required us to improve the performance of the tool itself: for example, for complex arithmetic and array operations. There is a substantial difference between completing a SAW proof and running it reliably under a particular time budget.

Running in CI means that proof failures are blockers on deployment. This is a pain point for engineering teams: if a proof blocks a release without good reason, the result is likely to be that the proof is disabled. To mitigate this, SAW makes extensive use of solver-backed automation, and we design our proofs to resist change. In many cases, small changes to the code result in no proof changes, which decreases the false-positive rate and increases the utility of the proof. However, some small changes still require significant proof modifications. In some cases, minor code changes actually require significantly different reasoning: for example, in code computing mathematical functions. In other cases, the main culprit is the solver-backed automation itself, which can be highly sensitive to minor changes in logical terms. The fragility of proofs under change is a known problem in many verification tools and the subject of ongoing research into proof-repair technologies.

In our experience, change-resistant proofs consist of layered specifications from low abstraction to high. The lowest level of abstraction should be the code, and the highest level of abstraction should be desirable properties about the code. In general, higher layers will correspond to increased proof effort. This is advantageous because the lowest level of abstraction is the most likely to experience change. In many cases, changes to code require only repairs to the lowest-level proofs without requiring the change to bubble all of the way up the stack. In SAW, the majority of these low-level changes can be discharged using SMT-backed automation.

Typically, SAW proofs require a small number of layers. Often, we use just two: a low-level proof from the implementation to a proved-safe abstraction and a high-level proof from this abstraction to the functional correctness property. As a larger example, our proof of the s2n HMAC primitive[1] was structured into four layers. The highest layer was a preexisting proof indistinguishable from randomness, the fundamental correctness property. This proof was completed using the Coq proof assistant and required extensive manual effort. Lower layers were verified mostly using SAW, with small amounts of Coq. The lowest layer connected to the implementation was verified using SAW exclusively to maximize automation.

## Proof Engineering Strategies

When building a proof, our most important strategy is to leverage automation wherever possible. The SMT-backed automation in SAW means that most of the trivial reasoning can be eliminated, which leaves us able to focus on legitimately difficult verification problems. We also avoid introducing internal specifications wherever possible as these are typically the most difficult to write and the most fragile to code change. Internal specifications are sometimes needed for scalability reasons, but by limiting them to the largest scope possible, we reduce effort and increase proof resilience.

Another important strategy we use is to separate concerns in the proof. In our cryptographic proofs, it is often possible to verify safety (for example, the absence of memory errors) without verifying functional correctness. We have found that much of the technical risk of the project exists at this stage. Focusing on this problem early in the project allows us to reduce risk. Separating these concerns also results in proof terms that are smaller and easier to understand and helps isolate failures when developing proofs of functional correctness.

When developing proofs, we have learned that most proof engineering time is spent working with a failing proof. Either the proof is in an incomplete state as it is constructed, or more rarely, the target system itself is incorrect. In either case, proof engineers must be able to inspect proof terms to diagnose errors, and they must be able to rapidly rerun updated versions of the proof. In software engineering, it is obvious that development requires working with incomplete software, but this perspective is less common in formal verification research. SAW itself provides some support for failing proofs, but there remains a lot of room for improved tools and practices.

## Proofs Need Proof Engineers

The type of verification Galois does with SAW focuses on a system's most complex and intricate properties. As a result, these proofs must be manually constructed by proof engineers. This is in contrast to fully automated tools, such as Facebook's Infer,[10] which nonexperts can use. Automated tools have proved their value at scale, but they are inherently limited to simpler properties of the system. For example, Infer can verify memory safety but only for relatively simple memory usage patterns. The type of labor-intensive verification that we practice is necessary to verify correctness properties for complex, highly optimized code. Our work has focused on cryptographic modules, but the same is true (for example) of core operating system components, as demonstrated by the SeL4 project.[7]

Proof engineering is analogous to software engineering in that it requires experienced teams with knowledge built up over multiple projects. A team must understand the system to be verified, which may be as difficult as writing the code in the first place. The team members must deeply understand the strengths and limitations of the proof tools and be able to modify them, if necessary. And they must be able to design proofs to be resilient to change, fit into CI budgets, and match the required assurance properties.

Galois's team has operated for five years with mostly consistent members over a variety of proof projects, both commercial and U.S. government. Over this time, we have refined our tools and working practices and achieved a dramatic increase in the scale of proofs that we can accomplish. Compared to our early engagements, such as the s2n HMAC proof, our most recent proof of the blst library was an order of magnitude more complex in code size, specification complexity, degree of optimization, and range of coding patterns. The increased effectiveness we have achieved does not come from any single innovation. Instead, we have collected many small improvements in our tools and strategies, and these improvements have compounded together over time.

Teams with this level of expertise are at present uncommon, although the number is growing year by year. This limits the reach of formal verification: scalability will require teaching many more people to engineer proofs. To some extent, we can solve this problem through more usable and better automated tools, which would lower the skill ceiling for many tasks. We are also optimistic that many proof breakages can be eliminated by building change-resistant proofs and through proof-repair techniques. We can also build tools that fit with the existing metaphors used by engineers: the CBMC tool[11] allows users to write "symbolic tests" that resemble standard test harnesses. We hope these improvements can democratize the process of proof engineering, making it accessible to many more engineers. However, we expect that there will always be some tasks that require specialist teams, just as there are in software engineering more generally.

The last five years have seen formal verification flourish in diverse industry niches. Our work on cryptographic verification has been a small but important part of this growing acceptance. The problems we face now as a field are familiar to any nascent technology. How can we make our approach better address the needs of our users? How can we make our tools more scalable, reproducible, and understandable? How do we articulate the value of our work in convincing ways?

Galois has made significant gains on these problems over the last five years. Our work shows the improvements that are possible for a single team learning and developing over several verification projects. We hope to now see a virtuous cycle where more teams adopt formal verification, which leads to more capable tools and more effective practices.

We see the potential for innovative research in several areas of proof engineering. Current proofs are fragile in the face of change; we need techniques that can resist and repair failures when they occur. Current tools are oriented around checking completed proofs; we need engineering-oriented tools that assist during proof construction and refactoring. Current proof projects are rather unpredictable in terms of cost and benefit; we need to make them a reliable and quantifiable part of software assurance. Current proof engineering requires deep expertise of the type that Galois's team has built over many years; we need to make proof engineering accessible to many more people.

The last five years have demonstrated that formal verification is ready to solve some of the most challenging assurance problems that industry has to offer. Now, we look forward to a world where formal verification takes its place as a standard assurance tool in industry. ■


### Acknowledgments

This article is a product of many conversations with the Galois team working on industry verification. Thanks to Matt Bauer, Brett Boston, Samuel Breese, Charisee Chiw, Andrey Chudnov, Nathan Collins, Joey Dodds, Brian Huffman, Ajay Kumar Eeralla, Giuliano Losa, Stephen Magill, Eric Mertens, Eric Mullen, Andrei Stefanescu, Mark Saaltink, Aaron Tomb, and Eddy Westbrook. Thanks also to the many Galwegians who have contributed to the SAW and Cryptol projects over two decades.

## References

1. J. Dodds *et al.*, "Continuous formal verification of Amazon s2n," in *Computer Aided Verification*, vol 10982, H. Chockler and G. Weissenbacher, Eds. Cham: Springer-Verlag, 2018, pp. 430–446.
2. B. Boston *et al.*, "Verified cryptographic code for everybody," in *Computer Aided Verification*, vol. 12759, A. Silva and K. R. M. Leino, Eds. Cham: Springer-Verlag, 2021, pp. 645–668.
3. J. Dodds. "Announcing the 'blst' BLS verification project." Galois.com. https://galois.com/blog/2020/09/announcing-the-blst-bls-verification-project
4. "The software analysis workbench." SAW.Galois.com. https://saw.galois.com/ (Accessed: Mar. 3, 2022).
5. "Cryptol." Cryptol.Galois.com. https://cryptol.net (Accessed: Mar. 3, 2022).
6. CompCert. https://compcert.org (Accessed: Mar. 3, 2022).
7. "The SeL4 Microkernel." SeL4. https://sel4.systems (Accessed: Mar. 3, 2022).
8. "Project Everest: Provably secure communication software." Project Everest. https://project-everest.github.io (Accessed: Mar. 3, 2022).
9. P. W. O'Hearn, "Continuous reasoning: Scaling the impact of formal methods," in *Proc. 33rd Annu. ACM/IEEE Symp. Logic Comput. Sci. (LICS '18)*, pp. 13–25, doi: 10.1145/3209108.3209109.
10. "A tool to detect bugs in Java and C/C++/Objective-C code before it ships." fbinfer.com. https://fbinfer.com (Accessed: Mar. 3, 2022).
11. "CBMC: Bounded model checking for software." cprover.org. http://www.cprover.org/cbmc/ (Accessed: Mar. 3, 2022).


**Mike Dodds** is a principal scientist at Galois Inc, Portland, Oregon, 97204, USA. His research interests include applying formal methods tools in industry, with a particular focus on cryptographic, concurrent, and distributed systems. Dodds received a Ph.D. from the University of York, U.K. Contact him at miked@galois.com.



*Digital Object Identifier 10.1109/MSEC.2022.3171997*