Verifying Security: Where We Are and Where We Should Go

Paul Vines

Jun 10, 2016

Abstract

Verification techniques are increasingly being applied to the area of computer security. However, they vary greatly in the tpyes of approaches and areas of security they investigate. In this paper, we attempt to systematize the current state of verification in security and provide guidance to where research effort could be most effectively be spent in the future.

1 Introduction

There are many benefits of applying verification to programs, security is a large one. Security can be a difficult property to ensure in programs for several reasons. First and foremost, it is not actually essential to the function of most programs. As such, many programs can 'get away' with failing to implement security properly, and no one will notice until much later. This means security does not have the same level of incentive to be done right that other aspects of most programs have, such as liveness or stability.

Secondly, properties guaranteed by security, and the operations required to implement those properties, can often be very subtle. In many cases security cannot be determined to fail until it is attacked, so bugs can creep in and be very difficult to detect in the development process. This is similar to bugs emergent from the complexity of distributed systems, where even extensive testing is often inadequate to find all problems[19, 15].

For these reasons, security is a potentially excellent area for applying verification techniques. In this work, we survey the existing applications of verification to the broadly-defined area of computer security. We provide a classification of current works to aid in understanding the variety of different approaches that have been taken and attempt to provide direction for where future research efforts should be focused.

2 Overview of Projects

There exist a handful of forays into verification applied to security [11, 16, 9, 5, 2, 14]. These projects span a wide variety of areas of what could be considered 'computer security', ranging from verification of implementations from functional specifications, such as those provided by FIPS, to proving cryptographic properties are provided by a particular functional specification assuming correct implementations, and projects combining both to provide full-stack verification from cryptographic proof to assembly instructions. To help breakdown the types of research that has been done, we provide the following systemization scheme.

2.1 Cryptographic Primitives

Projects in this space seek to provide verified implementations of the types of cryptographic primitives that serve as the basis for secure protocols. Examples of these primitives include secure hash algorithms (e.g. SHA256) and symmetric and asymmetric encryption algorithms (e.g. AES and RSA). These primitives are the basic blocks upon which more complex security protocols (see below) are built to facilitate real-world applications. The line between a primitive and a higher level protocol is not exact; for example, the HMAC algorithm that provides the ability to create a keyed message-authentication-code is requires a secure hash algorithm to function, but we consider HMAC itself to also be a cryptographic primitive both due to its small implementation complexity and the reliance on it by virtually all secure communication protocols.

Unlike some projects, research efforts to verify cryptographic primitives are clearly examples of verification applied to security. So far there have been several major efforts.

2.1.1 Verified RSA [2]

The first is a verification of an implementation of the RSA asymmetric encryption algorithm. This project

used a cryptographic verification framework called EasyCrypt [8] (see below) to enable verification of the cryptographic properties the RSA algorithm is supposed to provide. From this cryptographic specification, a functional specification of the algorithm is created and proved to provide the same cryptographic properties. This functional specification is then extracted to C code, which is verifiably compiled to machine instructions using the CompCert verified compiler [17]. In this project the extraction step is performed by an unverified Python script [9].

The Trusted Computing Base (TCB) of this implementation includes the EasyCrypt Framework, which relies on Why3 and SMT solvers, as well as its own set of axiomatic rules for facilitating cryptographic proofs, and the Ocaml runtime. Extracting C-code to input into CompCert from EasyCrypt's proven version of the algorithm also requires trusting the Python extraction code.

2.1.2 Verified SHA256 [5]

In addition to encryption algorithms, secure hashing algorithms are an essentialy part of security. One of the fundamental properties required of a secure hash algorithm is collision resistance. A collision resistant hash algorithm has no way to find two inputs such that the output of hashing them is equal, besides brute-force trying all possible inputs. Unfortunately, this property cannot actually be proven for currently used secure hash algorithms. This unprovability limits the usefulness of verifying an implementation of a secure hash function starting at the cryptographic properties, since some of them must be assumed. Nonetheless, a project used the Foundational Cryptography Framework [18], build atop Coq [1], to create an assembly implementation of the SHA256 algorithm [5]. This was produced by proving properties in FCF to provide a functional specification in Coq, then Verifiable C [4] was used to provide a proof of equivalence between this Coq specification and a C implementation, which was then compiled using CompCert.

The TCB of this implementation is the TCB of CompCert and Coq's proof checker.

2.1.3 Verified HMAC [9]

Following on the above work to create a verified implementation of SHA256, a verified implementation of the HMAC algorithm was also created. The HMAC algorithm provides a keyed message-authentication-code (MAC) that allows the receiver of a message to verify the message was not tampered with, if the receiver and sender both possess the same key and the sender applied the HMAC algorithm to the message. The methodology used in this proof was the same as that of the above SHA256 verification effort, except the proof of HMAC's properties must also rely on the proof of SHA256, since HMAC uses SHA256 as a part of its implementation.

2.1.4 Functional Verification of Cryptographic Primitives [16]

The Ironclad project took a different approach to verifying an entire suite of cryptographic primitives required as dependencies to provide verified secure communication. Specifically, it included implementations of SHA, HMAC, and RSA. The fundamental goal of the approach taken by Ironclad is different, however, in that they translated the FIPS functional specification of these cryptographic algorithms into their verificaton language, Dafny, and then proved these specifications were properly implemented. In essense, this design trusts the previous efforts that created the FIPS specification from desired cryptographic properties, as well as the translation of that FIPS specification into the verification specification in Dafny.

The TCB of Ironclad includes the the assembler and linker for the compiled machine code, and the verifier to check the correctness of BoogieX86 code.

2.1.5 Verified Timing Channel Resistance [7]

The above efforts have in common that they attempt to prove the correctness of an implementation of cryptographic primitives, to ensure the implementation provides certain cryptographic properties (in the case of Ironclad, based upon the assumption that the FIPS spec provides these properties). However, there are types of attacks against cryptographic algorithms that are not always well-captured by the original proofs. Namely, side-channel attacks compromise security properties of an algorithm by gaining additional information not considered in the attacker model. In the case of a timing side-channel, this means an attacker that can detect differences in timing of the execution of an algorithm, may be able to clean extra information about that algorithm's functioning, such as which command-flow branches were taken, or the nature of values stored in the cache.

Defending against side-channels is an active area of research outside of verification, but most of these approaches are not principled in how they are applied, and do not contain any verifiable guarantees about defeating side-channel attacks. A recent project [7] has provided a methodology for providing timing-channel mitigation. It does this my proving noninterference between the secret portions of a cryptographic algorithm implementation, such as the value of the key involved, with other por-

tions of the implementation that could be observable by an attacker, such as when caches or memory is accessed. They integrate this analysis as a new layer in the CompCert verified compiler to provide these guarantees in the instructions generated by the compilation of the program.

2.2 Security Protocols

While cryptographic primitives are essential components of computer security, in most cases a single primitive in isolation cannot be used to achieve the goal that is desired. For example, to securely communicate across a network while maintaining confidentialy, integrity, and authenticity of the messages between two devices, no single cryptographic primitive will work. Furthermore, most protocols require much more state, compatability negotiations, and myriad other considerations to actually be practically useful. Thus, security protocols have been created to facilitate these real-world uses, and leverage the various cryptographic primitives we have discussed above in order to provide strong security. Perhaps the most prevalent example of this is the Transport-Layer-Security protocol (TLS) that provides security for communications and is used by many applications, such as the HTTPS protocol.

2.2.1 miTLS [11]

TLS is a large and complex protocol, encompassing authenticating the server and client is connecting to, negotiating which security algorithms to be used, securely exchanging keys, establishing a secure channel, transmitting encrypted data between hosts, and securely ending the channel. There have been many vulnerabilities discovered in the TLS protocol (and its predecessor, SSL) over the years, ranging from flaws in the cryptographic-level specification down to the varied implementations of the protocol [10].

miTLS is an effort to create a verified implementation of TLS, working from the cryptographic properties the protocol is intended to provide. Technologically, miTLS uses a refinement typing system, F7, to implement the TLS protocol and ensure it probabalistically meets the secrecy requirements. This combination of F7 and F# code is ultimately compiled into .NET bytecode that is then run by the .NET runtime. Unlike the cryptographic primitive projects described above, miTLS has many trusted components below its implementation: the .NET runtime itself, as well as the .NET implementations of the cryptographic primitives it calls. The verification of miTLS itself revealed a new vulnerability built into the TLS specification. Additional work utilizing miTLS in subsequent years revealed additional new specification vulnerabilities [12] as well as a variety of errors in other implementations of the protocol [10].

The TCB of miTLS is large for a verification project, it includes the F7 typechecker, F# compiler, .NET runtime, and cryptographic libraries in .NET.

2.2.2 Verified SSH [14]

Another project verified an implementation of the Secure Shell protol (SSH), using a different cryptographic verification framework, CryptoVerif [13]. CryptoVerif provides a large degree of automation for proving the security properties of the SSH protocol. However, the project as a whole has a large TCB, including the CryptoVerif framework, a compiler from it to Ocaml, the Ocaml runtime, and all the cryptographic primitives used in the protocol.

2.3 Secure Applications

Above we discussed the two clearest applications of verification to security: verifying cryptographic primitives and verifying security protocols built on those primitives. Defining what constitutes verification of secure applications is more difficult. The line between verifying an app is secure and verifying an app functions correctly is essentially just a question of how one defines bugs and whether or not an app attempts to specify security properties for itself or not. In some sense, any application that is verified and has security concerns is an application of verification to security, because by verifying the app is functionally correct it is also verifying the app is avoiding some class of vulnerabilities that arise from being functionally incorrect. Thus, we limit our analysis of this space to a single example, Ironclad, although almost every other project that provides verification for an application is inherently also providing verification of security properties of some type as well.

2.3.1 Ironclad

As mentioned above, Ironclad provides verification of the implementation of several cryptographic primitives from the level of the FIPS specification down to assembly code. Here we discuss that it also provides verification of several example applications, such as a notary and a privacy-preserving database. Ironclad approaches verifying application security from the perspective of secret and public data. By default, nothing that is defined as secret should ever observably affect the output of an application (noninterference). From there, applications can define declassi-

fication policies that allow specific secret-influenced outputs to be released.

3 Value of Verifying Primitives

Cryptographic primitives represent an attractive target for applying verification to the security space; they are relatively small, they involve proof techniques unique to their domain (cryptography), and they form the basis for all other security software. Despite these attractive qualities, it is not clear that this is actually the most useful place to apply verification effort.

Because cryptographic primitives are small and simple, at least in terms of state, they are fairly easy to implement. Furthermore, because they are so important, software developers implementing them tend to do so very carefully, and a few well-written and well-tested implementations of these critical but small algorithms are reused. Finally, the fact that they require different proof techniques to be supported by verification tools makes them attractive as research contributions. However, it also means that developing tools for these proof techniques is not necessarily beneficial to other areas of verification. The main caveat to this point is that clearly cryptographers developing these cryptographic primitives need some kind of mechanical proof assistant to help ensure their proofs are correct. If these various frameworks developed for full program verification, like EasyCrypt or FCF, represent the best available tools for this, then it is certainly beneficial to use them to at least generate the functional specifications of new cryptographic primitives.

In terms of real impact on the vulnerability of software, current engineering practices actually appear to be good enough at correctly implementing these cryptographic primitives and effort should be focused on more vulnerability-prone areas of security. A notable exception is in the research of verifiable timing-channel defenses. Recent unverified approaches to timing-channel defense have been shown to be error-prone [3], so verification in this area may prove helpful.

4 The Challenge of Verifying Secure Protocols

The greater complexity of trying to verify security protocols instead of cryptographic primitives can be seen in how much larger the trusted at-runtime TCB of miTLS is compared to, for example, verified HMAC which runs directly as instructons and does not require trusting external software. Furthermore, the fact that flaws were quickly found in this verified implementation of TLS suggests even verification approaches to improving security protocols is on unstable ground.

Cryptographic primitives represent pervasive dependencies for security, but are also overall low-risk in their current unverified state. Higher-level security protocols on the other hand are also widespread, but represent a much larger portion of vulnerabilities. Therefore, of the two categories, verification of security protocols like TLS represents a more impactful application of verification techniques.

Another interesting fact emerges from the saga of TLS vulnerability research and verification. An unverified implementation of TLS, MbedTLS (then called PolarSSL) [6] was found to be equally bug-free as miTLS when performing comparison-testing to find flaws in implementations of the protocol. MbedTLS does not use any verification techniques, but simply emphasizes simplicity and other good software engineering practices in its development. This provides evidence that, similar to the case in cryptographic primitives, perhaps full program verification is not actually the only solution to the plague of errors in security protocols. Perhaps verifications from cryptographic properties to a functional specification of these protocols, followed by a well-disciplined implementation, is actually similarly effective. In the short-term, while verification is still decidedly difficult and raises efficiency problems [11], it may make the most sense to proceed in as modular an approach as possible. This could also help provide incentives for academic research groups to work in this area, rather than working on the more tractable but less impactful projects verifying primitives.

5 Verifying Secure Applications

As noted in the overview of projects, verifying security of applications is inherent in many other verification projects. Given the current state of security and application verification it makes the most sense to focus efforts of security-specific verification at widely used protocols, like TLS, rather than specific applications.

6 Conclusion

Verification in security has certainly arrived as a research area, and recent projects have represented a relatively wide variety of approaches. The field, however, still seems focused on proving a certain area of security can have verification applied to it, rather than focusing on trying to produce useful artifacts for actual use. In par-

ticular, the focus on cryptographic primitives in several projects is contrary to what would be practically useful to obtain from verification efforts, because implementation errors in these primitives do not represent a problem area for security today. Our recommendation is that near-term verification efforts be focused on more complex, and vulnerability-prone, security protocols like TLS. However, development of new tools for aiding verification in all areas of security is probably a worthwhile longterm effort.

References

[1] The coq proof assistant.

[2] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1217–1230. ACM, 2013.

[3] J. B. Almeida12, M. Barbosa13, G. Barthe, and F. Dupressoir. Verifiable side-channel security of cryptographic implementations: constant-time mee-cbc.

[4] A. W. Appel. Verified software toolchain. In *Programming Languages and Systems*, pages 1–17. Springer, 2011.

[5] A. W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(2):7, 2015.

[6] ARM. mbedtls. 2015.

[7] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1267–1279. ACM, 2014.

[8] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub. Easycrypt: A tutorial. In *Foundations of Security Analysis and Design VII*, pages 146–166. Springer, 2014.

[9] L. Beringer, A. Petcher, Q. Y. Katherine, and A. W. Appel. Verified correctness and security of openssl hmac. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 207–221, 2015.

[10] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 535–552. IEEE, 2015.

[11] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing tls with verified cryptographic security. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 445–459. IEEE, 2013.

[12] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 98–113. IEEE, 2014.

[13] B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In *Advances in Cryptology-CRYPTO 2006*, pages 537–554. Springer, 2006.

[14] D. Cadé and B. Blanchet. From computationally-proved protocol specifications to implementations. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pages 65–74. IEEE, 2012.

[15] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

[16] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, 2014.

[17] X. Leroy. The compcert c verified compiler. *Documentation and users manual. INRIA Paris-Rocquencourt*, 2012.

[18] A. Petcher. *A Foundational Proof Framework for Cryptography*. PhD thesis, 2015.

[19] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 357–368. ACM, 2015.