Applying Formal Methods to Secure the Software Supply Chain

Philip Johnson-Freyd*and Jon Aytac†and Geoffrey Hulette‡

May 26, 2021

Response to bullet 3: Initial list of secure software development lifecycle standards, best practices, and other guidelines acceptable for the development of software for purchase by the federal government. This list of standards shall include criteria and required information for attestation of conformity by developers and suppliers. See EO Section 4(e)(i, ii, ix, and x).

Formal methods-based static analyses of program source code are able to provide a very high level of assurance that a software program meets critical safety and security requirements. In particular, formal methods can detect errors and vulnerabilities that traditional verification techniques (e.g., testing or linting) alone will not [9].

Testing can determine only how a software program behaves on certain given inputs. For nontrivial programs, testing each and every possible input is usually infeasible because the number of potential inputs is very large. Therefore, although testing may find unsafe or insecure program behaviors, it cannot rule them out. Linting techniques analyze the program text (i.e., source code) and are useful for drawing a programmer's attention to likely problems therein but, like testing, linting cannot rule out errors or security violations.

Formal methods, like linting, work by analyzing program source code. Unlike linting, formal methods consider program semantics in order to prove, in the sense that one proves a mathematical theorem, that given semantic properties are met. Properties that can be checked include so-called "safety properties," such as the absence of runtime errors (e.g., dereferencing an invalid memory address or executing a division by zero) [4]. Properties may also express invulnerability to a class of exploits [5, 3]. Furthermore, properties can be combined to express complex notions of full system correctness.

Formal methods have been shown to be effective at detecting security vulnerabilities in practice [5]. At Sandia we have observed that the process of formal analysis can increase safety and reliability of the resulting system beyond even what is explicitly specified. For instance, VST [1] detects common sources of security vulnerabilities in C code, such as undefined behavior and runtime crashes, whether explicitly specified or not. Similarly, "lightweight" formal methods, such as strong type systems and model checking at the design level, reduce bugs in practice [6].

However, the greatest potential benefit of formal methods will be facilitating use of a Trusted Computing Base (TCB) [3] as a foundation for security. In this scheme, formal methods are used to prove that a given program is secure, obviating the need to trust it so long as its specification, as well as the tools necessary to check its specification, are available and trusted. Variations on this idea, such as proof carrying code [10], suggest an approach to software security based on the principle of minimizing the need for trust.

Proof assistants based on the de Bruijn criterion [2, 11] feature a small, simple proof checking kernel that is independent from the complex facilities for finding proofs. Because the kernel is so small it can be subjected to extensive human-audited trust evaluation, and used as the foundation of the TCB, minimizing the critical surface area of the overall system. Methods based on foundational proof assistants satisfying this criterion should therefore be preferred whenever possible.

For software to be trusted, its entire development toolchain must be considered. Even if source code is correct, compilers, assemblers, and linkers may potentially introduce vulnerabilities and so must be trusted. Compilers such as CompCert [8] and CakeML [7] are themselves proven correct in foundational proof assistants and thus can be removed from the TCB. However, gaps remain. The compilers themselves must be

*pajohn@sandia.gov
†jmaytac@sandia.gov
‡ghulett@sandia.gov

compiled, as must the proof assistant, so trusted "extraction" mechanisms are needed. Even with such tools, the classic problem of "trusting trust" [12] (that the verified compiler was derived via a compilation chain from an unverified one) is still a point of concern. To mitigate this issue we advocate the use of multiple independent implementations of the proof checking kernel and whatever other minimal machinery is needed to bootstrap a verified software toolchain.

Ultimately, formal methods-based verification, including of development tools, and based on a minimal trusted computing base, provides the highest possible level of assurance for software. As such, it should be considered the gold standard and the goal for all high consequence systems. For the most safety- and security-critical software, resources should be deployed towards building more complete formal methods-based verification stacks, more trustworthy formal methods tools, and ever smaller Trusted Computing Bases.

Authors

Philip Johnson-Freyd	Jon Aytac	Geoffrey Hulette
Senior Member of Technical Staff	Senior Member of Technical Staff	Principal Member of Technical Staff
Sandia National Laboratories	Sandia National Laboratories	Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.
SAND2021-6433 O

References

[1] Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, page 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, 2002.

[3] Giampaolo Bella. *Formal correctness of security protocols*. Springer Science & Business Media, 2007.

[4] Patrick Cousot. Proving the absence of run-time errors in safety-critical avionics code. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 7–9, 2007.

[5] Kathleen Fisher, John Launchbury, and Raymond Richards. The HACMS program: Using formal methods to eliminate exploitable bugs. *Philosophical Transactions of The Royal Society A Mathematical Physical and Engineering Sciences*, 375:20150401, 10 2017.

[6] Daniel Jackson. Lightweight formal methods. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, pages 1–1, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[7] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices*, 49(1):179–191, 2014.

[8] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert - a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[9] Jackson R. Mayo, Robert C. Armstrong, and Geoffrey C. Hulette. Digital system robustness via design constraints: The lesson of formal methods. In *2015 Annual IEEE Systems Conference (SysCon) Proceedings*, pages 109–114, 2015.

[10] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106–119, New York, NY, USA, 1997. Association for Computing Machinery.

[11] Martijn Oostdijk and Herman Geuvers. Proof by computation in the Coq system. *Theoretical Computer Science*, 272(1-2):293–314, 2002.

[12] Ken Thompson. Reflections on trusting trust. In *ACM Turing award lectures*, page 1983. ACM, 2007.