

Assurance and Verification of Security Properties

Constance Heitmeyer^{*}
Naval Research Laboratory
Washington, DC 20375
constance.heimtaylor@nrl.navy.mil

ABSTRACT

This paper begins by reviewing our earlier research in assurance and verification of embedded software systems. This research was part of an effort in which we prepared evidence, including an abstract formal model and mechanical proofs, of the security of an embedded system implemented in the C language. It then describes the evidence provided for the certification and how this evidence might be presented in an assurance case. The paper also describes our more recent research on SecProve, a tool for automatically checking the security of C programs during their development. It concludes by listing some open research problems in system assurance and verification.

1. INTRODUCTION

Critically needed are new methods for obtaining high assurance systems, i.e., systems that satisfy their requirements. One effective approach to building high assurance systems is to build a formal model of the system's required behavior and then to apply formal verification to demonstrate that the system model satisfies critical properties, such as security properties. However, in developing high assurance systems, formal models and proofs are insufficient. Other evidence is needed, e.g., validation of the system requirements and software tests, to obtain high assurance that the system behaves as intended.

Section 2 reviews our experience preparing evidence supporting the certification, based on the Common Criteria, of the security of an embedded software system [9, 8]. Section 3 discusses how the evidence developed for CD's certification could be used in an assurance case, while Section 4 introduces our recent research in assurance and verification. In this research, we are developing a tool for automatically verifying the security of embedded systems implemented in the C language.

^{*}This research is supported by the Office of Naval Research.

2. HIGH ASSURANCE OF SECURITY

2.1 The Embedded System (CD)

CD is an embedded software system which processes data in a computer whose memory has been divided into non-overlapping partitions. While at any given time the data stored and processed by CD in one memory partition is classified at a single security level, CD may later reconfigure that partition to store and process data at a different security level. Because CD stores and processes data classified at different security levels, security violations by CD, e.g., "leakage" of data from one memory partition to another, could cause significant damage to national security.

To prevent violations of *data separation*, e.g., the leakage of data from one memory partition to another, CD uses a *separation kernel* to mediate every access to memory. By definition, a *separation kernel* [17] mimics the separation of a system into a set of independent virtual machines by dividing the system memory into partitions and restricting the information flow between the partitions. By mediating every access to memory, CD's separation kernel ensures that every memory access is authorized and that every transfer of data from one CD memory location to another is authorized. Any attempted memory access by CD that is unauthorized causes an exception.

As noted above, CD's certification was based on the Common Criteria. A number of international organizations established the Common Criteria to provide a single basis for evaluating the security of information systems citeCC2004. Associated with the Common Criteria are seven Evaluation Assurance Levels. EAL7, the highest assurance level, requires a formal specification of a system's security functions and security model, and formal proof of correspondence between the two.

2.2 Approach

The Common Criteria certification required evidence that CD satisfies data separation. To produce this evidence, we needed to specify the security properties that CD was required to satisfy, formally model CD's security functions, and specify the behavior of the C code that implemented CD. In contrast to Klein et al. [10], who formally verified the full functional correctness of seL4, a general-purpose operating system microkernel, we undertook an ambitious, albeit simpler task—formal verification that the code implementing CD's separation kernel enforced data separation. This was achieved by proving security properties such as

no-exfiltration and temporal separation. (For details, see [8, 9].) The means chosen to specify CD’s code behavior was annotation of the code with Floyd-Hoare pre- and postconditions. The process we formulated for establishing that the annotated code satisfies the security properties contained the following five steps:

- A. Formulate a Top Level Specification (TLS) of the code as a state machine model. The TLS is CD’s required security model. Formally express the required security properties as properties of the state machine model.
- B. Translate the TLS and the properties into the language of a mechanical prover, and prove formally that the TLS satisfies the properties.
- C. Implement the C source code implementation.
- D. Annotate the source code with pre- and postconditions.
- E. Demonstrate that the C code satisfies the TLS.

An initial assumption was that the five-step process described above would proceed top down. Such a top-down design has in fact been followed by others; for example, Klein et al. formulated “an abstract specification” for the C code implementing seL4 which precisely describes argument formats, encodings, and error reporting required at the C code level; for instance, some C-level size restrictions are visible at this level. However, our analog to this abstract specification, the TLS, abstracts away detail, such as argument formats, encoding and error reporting, and represents memory areas with symbolic names and the contents of memory as integers. Hence, there was a wide gap between our (abstract) TLS of CD’s kernel and the (concrete) annotated C code that implements the kernel.

2.3 Showing Code-to-TLS Conformance

Because the TLS could not be formulated until the C code was written, our plan to follow a top-down process was infeasible. However, once the C code was implemented, the TLS and the properties were rapidly formulated, the proof that the TLS satisfied the properties was established, and the process of annotating the C code with pre- and postconditions began.

Because the properties to be proved were all preserved by refinement, only one major step remained: showing that the annotated C code (the Actual Code Behavior) satisfied the TLS. To solve this problem, we needed an additional step between D and E (step D1). In step D1, we needed to specify the Desired Code Behavior, i.e., relate the abstract states and state transitions in the TLS to the concrete states and state transitions in the C code and the pre- and postconditions in the TLS to the pre- and postconditions that annotated the C code. To establish correspondence between concrete states in the code and abstract states in the TLS, a function was defined which relates concrete entities, such as memory areas, code variables, and logical variables, to abstract state variables in the TLS, such as memory areas

of interest (MAIs)¹, and mapping the value space of each concrete entity to that of its corresponding abstract state variable. For example, the actual physical addresses of the MAIs are mapped to their corresponding abstract state variables, i.e., the symbolic names of memory areas, in the TLS. A second map relates assertions (i.e., pre- and postconditions) at the abstract TLS level to equivalent assertions at the code level.

2.4 Required Effort and Time

Formulating the TLS and the security properties required only 2.5 weeks, while translating the TLS and the properties into the mechanical prover PVS [18] and using PVS interactively to verify the properties required about 3.5 weeks. Defining the Desired Code Behavior required approximately five weeks. In contrast, annotating the source code and defining the requirements (needed before the TLS could be formulated) took much longer. For example, manual annotation of the C code required several months.

3. AN ASSURANCE CASE FOR CD

By definition, an *assurance case* is “a documented body of evidence that provides a convincing and valid argument that a specified set of critical claims about a system’s properties are adequately justified for a given application in a given environment” [12]. An assurance case is composed of *claims*, *argument*, and *evidence*, where the argument justifies the claims (in our case, about CD’s security) based on evidence about the system and its environment. Although an assurance case for CD was never formulated, based on the evidence developed for CD’s certification, we describe how one might formulate an assurance case for CD.

3.1 Claim

The major claim is that CD’s implementation enforces data separation.

3.2 Evidence

The following evidence was prepared and presented to the certifiers to support the claim that CD enforces data separation:

- The TLS, statement of properties, and proofs that the TLS satisfies the properties
- Annotated C code
- Desired Code Behavior and demonstration of correspondence between the annotated code and the Desired Code Behavior
- Validation of the TLS, including assumptions about CD’s environment, and of the security properties.
- Walk-through of the annotated kernel code
- Results of software tests
- Replay of interactive proofs

¹By definition, a *memory area of interest* is a memory area containing data whose leakage would violate data separation.

Formal Evidence. A significant component of the evidence is the set of artifacts we developed during the process outlined in Section 2. These artifacts include the formal security model (i.e., the TLS), the statement of the security properties that define data separation, the mechanized proof that the model satisfies the properties, the annotated C code, the formulation of the Desired Code Behavior, and demonstration that the Actual Code Behavior (i.e., the annotated code) corresponds to the Desired Code Behavior.

Validation. The TLS’s formulation of the kernel’s required behavior was represented as a state machine model both informally and formally. The informal version of the TLC was precisely expressed in natural language in the style of [11, 14]. This natural language version made it easy for the certifiers and the software developers of CD to validate the model. Similarly, the required security properties were also expressed in natural language. Review of the model detected a few minor errors. In contrast, review of the properties revealed that satisfying the temporal separation property was infeasible. Hence the statement of this property was weakened and the property reformulated. Review of the assumptions in the TLS also proved useful. Conflicting views on the assumptions were expressed. Eventually some assumptions were changed, a few in significant ways, to resolve the conflicting views.

3.3 Argument

The argument justifying the claim about CD’s security was only partial. Two reports were delivered to the certifier describing the relationship between the TLS, the properties, the proofs that the TLS satisfied the properties, the Desired Code Behavior, the Actual Code Behavior (i.e., the annotated C code), and the demonstration of correspondence between the Desired and Actual Code Behavior; and how these pieces taken together justify the claim. However, no explicit argument was made regarding the other evidence (e.g., the software tests, the code walkthrough, etc.), how this evidence was related to the formal evidence, and how all of the evidence taken together shows that CD enforces data separation.

A major argument that was missing is why proving security properties about CD’s separation kernel implied that these security properties were satisfied by CD as a whole. In contrast to CD’s separation kernel whose implementation contained on the order of 3,000 lines of C code, CD as a whole contained significantly more code. Hence, verifying the security of CD in its entirety using formal modeling and formal verification was impractical. However, because CD’s security kernel is clearly an extremely important component in reasoning about CD’s security, in my view, a strong argument can be made that showing that the CD kernel is secure implies that CD as a whole is secure.

4. AUTOMATING CODE VERIFICATION

As stated above, one activity that required significant time to complete was manual annotation of the C code. To make code verification practical, more automation is needed. To address this problem, we are developing a new tool called SecProve, which can automatically check C code for application-specific security properties [2, 5, 13]. Automated reasoning in SecProve is based on code assertions generated from three sources—assertions in the C code, property statements, and

the code itself—and used both in generating assertions and checking the validity of assertions with a theorem prover.

4.1 Overview of SecProve

The objective of SecProve is to automatically check the validity of security properties in a C program under development by a software programmer. SecProve is designed to behave like commercial static analysis tools (e.g., Coverity [1]) which automatically detect code vulnerabilities in programs written in C and other source code languages. However, unlike static analysis tools which detect application-independent property violations such as buffer overflows, SecProve is designed to detect violations of application-dependent properties, such as data separation. Designed to be push-button and easy to use, SecProve is designed to provide an integrated development environment for developing a C program, specifying application-specific security properties, automatically analyzing the C program at any point in development for property violations, and providing feedback useful in diagnosing and repairing detected security violations. Moreover, using SecProve does not require theorem proving skills.

SecProve is designed to incorporate automated theorem proving support from the general purpose theorem prover PVS [18] and KTP [16], a prover under development at NRL that combines k-induction with calls to an SMT solver, and thus uses standard automated deduction techniques. SecProve’s analysis of a program is supported by assertions associated with the C code. The assertions fall into two classes: Class 1, known to be valid; and Class 2, desired to be valid, but requiring verification.

To achieve our goal of automatic analysis and its support through assertion generation, SecProve provides templates to facilitate the formal statement by the developer of the security properties; and restricts the subset of C for which it will perform analysis. The property templates make it possible to automatically generate code assertions needed to establish given application-specific security properties (e.g., data flow properties). Current restrictions on C are that all loops are of bounded size and therefore unwindable, there are no recursive function calls, there is no overlap between expressions and commands, and pointers can only be used for function parameters.

SecProve’s approach to analyzing C programs is similar to that used by other C program analysis tools: it restricts the allowable C code and translates the code into equivalent code in a high level intermediate language better suited to direct code analysis. This approach is also used by the CIL (C Intermediate Language) compiler [15] and VCC [6], a tool for verifying concurrent C programs. SecProve is a prototype being developed as a proof of concept. Partly for this reason, we use LEMA (Language and Environment for Modeling Algorithms) [3, 4], a prototype algorithm verification tool developed at NRL, as the intermediate language in SecProve. The existing LEMA tool support has at least two advantages: it includes theorem proving support from PVS [18] and is specifically designed to reason about code assertions. The LEMA techniques used in [3, 4] to prove functional correctness of algorithms are being extended in SecProve to prove security properties of code.

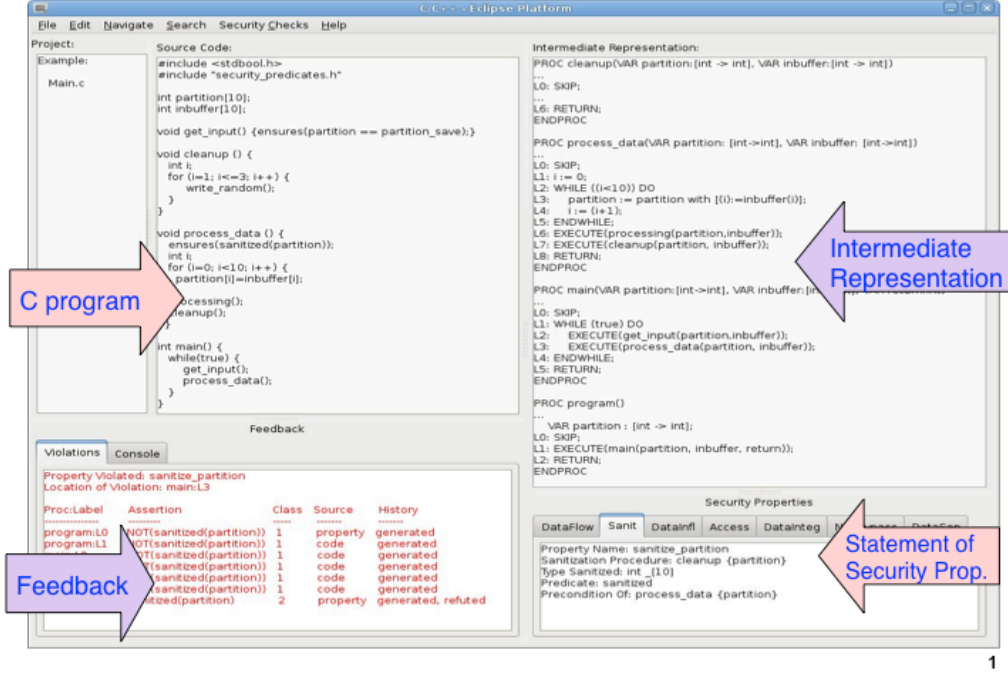


Figure 1: SecProve GUI.

The SecProve prototype is implemented as an Eclipse plugin, and the SecProve GUI, shown in Figure 1, is based on Eclipse. As shown in Figure 1, the GUI provides four windows: the Source Code and Security Properties windows are for user input, and the Intermediate Representation and Feedback windows are for user feedback. The Source Code window shows a partial C program containing only the declaration of a function, and missing both declarations and function definitions. The Security Properties window shows a completed template defining a sanitization property, an application-specific security property. The Intermediate Representation window shows the LEMA representation of the C program, edited to show the parts relevant to the sanitization property defined in the Security Properties window. Finally, the Feedback window reports that the C program violates the sanitization property and lists the violated code assertions.

4.2 Generating Assertions in SecProve

In SecProve, assertions may be generated in three ways: from developer-supplied assertions and contracts, from developer-defined properties, and from the C code. We discuss here our approach to generating assertions from C code. Most assertions are generated from LEMA code using forward propagation—**sp** computation. All assertions generated by forward propagation based solely on the code are Class 1 assertions. Except for assignment statements, forward propagation for the major LEMA code constructs is generally as straightforward as the backward propagation used in computing weakest preconditions or weakest liberal preconditions (**wp** or **wlp**). To date, we have developed techniques for generating assertions for four C constructs—assignment statements, conditional statements, while statements, and procedure calls [13].

5. ASSURANCE CASES WITH SECPROVE

In formulating an assurance case for SecProve, the major claims are the security properties defined by the developer. The evidence would include the statements displayed in the Feedback Window of the GUI, e.g., “Property Data.Flow is Valid”. A problem, however, is that currently SecProve does not provide proofs of properties—it simply tells the developer whether or not the property is violated. Moreover, because the security properties are stated at the level of the C code, validation of the security properties—i.e., that they are in fact the *intended* security properties—may be difficult. This is also a problem for the functional correctness proofs and proofs of security properties developed in verifying the operating system microkernel seL4 [10]. Because the abstract specification formulated in the seL4 formal verification was at the C code level, validating that the functional correctness property and the security properties proven about seL4, e.g., confidentiality and integrity, may not be as straightforward as in CD. Because CD’s TLS was at a high level of abstraction and because the properties were stated both in natural language and formally, validating that the properties captured CD’s intended behavior was easy.

6. OPEN RESEARCH ISSUES

Presented below are a number of open issues in assurance and verification that require further research.

6.1 Assurance Cases for Security

To date, most assurance cases have focused on safety properties. More research is needed to explore how assurance cases for security should be designed. What do assurance cases for security have in common with safety cases? In what ways are they different?

6.2 Further Research for SecProve

Listed below are several critical research questions which need to be explored before SecProve can be deployed.

- How can we automatically generate assertions for contracts and loop invariants (undecidable in general)?
- What are soundness arguments for our assertion techniques?
- How can we more fully automate assertion checking?
- How can we reduce the amount of information passed to a theorem prover during assertion checking?
- How can we extend SecProve's ability to handle pointers in C?

7. CONCLUSIONS

This paper reviewed the approach that we applied in preparing evidence for the certification of the security of an embedded software system called CD. This evidence included a formal model and formal verification that the model satisfied a set of security properties. The paper then presented an assurance case for the security of CD and discussed some limitations of this assurance case. The paper also introduced a new tool called SecProve which can be used by a developer to check security properties while he or she is writing a C program. The paper concludes by identifying a number of research issues in assurance and verification that need further study.

8. ACKNOWLEDGMENTS

The ideas in this paper have been developed jointly with my colleagues Myla Archer and Elizabeth Leonard. I gratefully acknowledge their substantive contributions to the research and to the definition of the open research problems.

9. REFERENCES

- [1] Coverity Prevent. Tool available at <http://coverity.com/>.
- [2] M. Archer, C. Heitmeyer, E. Leonard, C. Gasarch, and W. Ding. Automated tools for building secure software programs. U.S. Patent Application. Filed January 24, 2014 (pending).
- [3] M. Archer and E. Leonard. Establishing high confidence in code implementations of algorithms using formal verification of pseudocode. In *Proceedings, 3rd International Verification Workshop (VERIFY'06)*, 2006.
- [4] M. Archer, E. Leonard, and C. Gasarch. Verifying LEMA specifications in TAME. Technical Report NRL/MR/5540-12-9442, Wash., DC, 2013.
- [5] M. Archer, E. I. Leonard, and C. L. Heitmeyer. Idea: Writing secure C programs with SecProve. In *Engineering Secure Software and Systems - 5th Internat. Symp., ESSoS 2013*, volume 7781 of *Lecture Notes in Computer Science*, pages 171–180, Paris, FR, 2013. Springer.
- [6] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.
- [7] Common criteria for information technology security evaluation, Parts 1–3. Technical Report CCIMB-2004-01-001 through CCIMB-2004-01-003, Version 2.2, Revision 256, Jan. 2004.
- [8] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Applying formal methods to a certifiably secure software system. *IEEE Trans. Software Engineering*, 34(1):82–98, 2008.
- [9] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proc. 13th ACM Conf. on Comp. and Comm. Sec.*, 2006.
- [10] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, Feb. 2014.
- [11] C. E. Landwehr, C. L. Heitmeyer, and J. D. McLean. A security model for military message systems. *ACM Trans. Comput. Syst.*, 2(3):198–222, 1984.
- [12] Z. Langari and T. Maibaum. Safety cases: A review of challenges. In *Assurance Cases for Software-Intensive Systems (ASSURE)*, 2013 1st International Workshop on, pages 1–6, May 2013.
- [13] E. Leonard and M. Archer. Automated deduction techniques for reasoning about security properties of programs. Submitted for publication.
- [14] J. McLean, C. Landwehr, and C. Heitmeyer. A formal statement of the military message system security model. In *1984 IEEE Symposium on Security and Privacy Conference Proceedings*, pages 188–194, 1984.
- [15] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings, 11th Internat. Conf. on Compiler Construction, CC '02*, pages 213–228, 2002.
- [16] T. Nguyen, E. I. Leonard, M. M. Archer, and C. L. Heitmeyer. KTP: SMT-based k-induction theorem proving for state machine models. Draft report.
- [17] J. Rushby. Design and verification of secure systems. In *Proceedings, 8th ACM Symp. on Operating System Principles*, Dec. 1981.
- [18] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide, Version 2.4. Technical report, Computer Science Lab, SRI International, Menlo Park, CA, Nov. 2001.