Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 3–9
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

On the Role of Formal Methods in Software Certification: An Experience Report

Constance L. Heitmeyer[1,2]

Naval Research Laboratory
Washington, DC 20375

Abstract

This paper describes how formal methods were used to produce evidence in a certification, based on the Common Criteria, of a security-critical software system. The evidence included a top level specification (TLS) of the security-relevant software behavior, a formal statement of the required security properties, proofs that the specification satisfied the properties, and a demonstration that the source code, which had been annotated with preconditions and postconditions, was a refinement of the TLS. The paper also describes those aspects of our approach which were most effective and research that could significantly increase the effectiveness of formal methods in software certification.

Keywords: formal methods, security, software, formal verification, formal specification, certification

1 Introduction

Prior to its deployment, a security-critical software system may undergo a formal certification to demonstrate that it satisfies critical security properties. Although formal methods are part of the standard recommendations for developing and certifying security-critical systems, how to integrate formal methods into the certification process is, in large part, unclear. Especially challenging is how to demonstrate that the end product of software development—the code—behaves securely. This paper describes the formal methods and tools that our group applied to produce evidence for the certification, based on the Common Criteria, of a security-critical software system. It also describes the most effective aspects of our approach for certification and research that could significantly increase the utility of formal methods in software certification.

[1] Email:heitmeyer@itd.nrl.navy.mil

[2] The support of the Office of Naval Research is gratefully acknowledged. My NRL colleagues, Myla Archer, Elizabeth Leonard, and John McLean, contributed to this research.

doi:10.1016/j.entcs.2009.09.001

2 Background

A group of international organizations established the Common Criteria to provide a single basis for evaluating the security of information technology products [5]. Recently, our group prepared evidence to support the certification, based on the Common Criteria, of a security-critical, embedded software device called ED (Embedded Device). Required in the certification were 1) a formal proof of correspondence between a formal specification of ED's security functions and its required security properties *and* 2) a demonstration that ED's code satisfied the formal specification.

The device of interest, ED, processes data in an embedded system whose memory has been divided into non-overlapping partitions. Because it stores and processes data classified at different security levels, security violations by ED could cause significant damage. To prevent violations of *data separation*, e.g., the "leaking" of data from one memory partition to another, the ED design uses a *separation kernel* [16], a tamper-proof, non-bypassable program which mediates access to memory. By mediating every memory access, the kernel ensures that every access is authorized and that every transfer of data from one memory location to another is authorized. Any attempted memory access by ED that is unauthorized will cause an exception.

3 Our Approach to Certification

To provide a foundation for proving the security of ED, the code implementing the separation kernel was annotated with preconditions and postconditions in the style of Hoare and Floyd. The evidence we produced to demonstrate that ED enforces data separation included a Top Level Specification (TLS) of the separation-relevant behavior of the kernel, a formal statement of data separation, and a mechanized formal proof that the TLS satisfies data separation. In addition, we partitioned the annotated code into three categories, each requiring a different proof strategy. Finally, we established the formal correspondence between the annotated code and the TLS.

Given 1) source code annotated with preconditions and postconditions and 2) a security property of interest, the overall problem is how to establish that the code satisfies the property. We developed a five-step process for establishing the property. These five steps are described next.

3.1 Formulate a Top Level Specification

The purpose of the Top Level Specification (TLS) is to provide a precise, yet understandable description of the security-relevant behavior of the code and to make explicit the assumptions on which the security of the code is based. [3] In our approach, the TLS is represented in precise natural language as a state machine model, using the style of [12]. The advantage of precise natural language is that it enables stakeholders with differing backgrounds and objectives—the project manager, software

[3] For example, the assumptions for ED make explicit those routines that the certification authority agreed were outside the scope of the formal verification.

developers, evaluators, and the formal methods team—to communicate precisely about the required kernel behavior and helps ensure, early in the verification process, that misunderstandings are weeded out and issues resolved. Another purpose of the TLS is to provide a formal context and precise vocabulary for defining data separation. For the details of the TLS for the ED kernel, see [10].

3.2 Formally Represent the Security Properties

In our approach, the required security properties are formally expressed as properties of the state machine model that underlies the TLS. To enforce the required security property (data separation), ED must prevent data in a partition i from influencing or being influenced by 1) data in a partition j, where $i \neq j$, or 2) data in an earlier configuration of partition i. To formally represent data separation, we formulated a number of properties in precise natural language. Examples of the properties are `No-Exfiltration`, which states that data processing in any partition j cannot influence data stored outside the partition, and `No-Infiltration`, which states that data processing in any partition i is not influenced by data outside that partition.

3.3 Apply a Mechanical Prover

To demonstrate that the TLS satisfies the security properties of interest, the TLS and the properties are translated into the language of a theorem prover and the prover applied to prove formally that the TLS satisfies the properties. To formally verify that the TLS for the ED kernel enforces data separation, the natural language formulation of the TLS and the properties that represent data separation were translated into TAME (Timed Automata Modeling Environment) [3], a front-end to PVS [14], and TAME proofs were interactively constructed to show that the TAME specification satisfies each property.

3.4 Partition the Code

To show formally that the system is secure, we prove that the system code is a refinement of the state machine that underlies the TLS. For ED, our proof of refinement is based on a demonstration that all kernel code falls into three major categories: Event Code, Trusted Code, and Other Code (see [10] for details). Partitioning the code dramatically reduces the cost of code verification since only Event Code, a small part of the code, must be checked for conformance to the TLS. In ED, Event Code and Trusted Code comprised less than 10% of the code. The remaining 90% was Other Code.

3.5 Demonstrate Code Conformance

The final step is to show that each category of code is secure. To demonstrate that the kernel's Event Code is secure (i.e., does not violate data separation), we constructed two mappings: 1) a mapping from the Event Code to the TLS events

and from the code states to the states in the TLS, and 2) a mapping from preconditions and postconditions in the TLS events to the preconditions and postconditions that annotate the corresponding Event Code. We demonstrated separately that Trusted Code and Other Code were benign. Based on these results, we concluded that the kernel code refines the TLS. Because in step 3, we proved that the TLS satisfies the properties that guarantee data separation and because each property is preserved under refinement, we may conclude that the kernel code is secure. For details, see [10].

4 What Worked

4.1 Use of Natural Language

During certification, the natural language representation of the TLS enabled the evaluators to communicate easily with the formal methods team and others, thus ensuring that misunderstandings were avoided and issues resolved early in the certification process. The natural language representation of the TLS for ED contrasts with the representations used in other formal specifications of secure systems. These specifications are often expressed in specialized languages such as ACL-2 (see, e.g, [8]). Any ambiguity in the natural language representation of the kernel behavior was removed by translating the TLS into TAME, since the state machine semantics underlying TAME is expressed as a PVS theory.

4.2 Use of Scenarios to Understand Requirements

One significant challenge was to understand the security-relevant behavior of the ED kernel. To accomplish this, we designed several scenarios, i.e., sequences of events, and executed them using the SCR (Software Cost Reduction) simulator [11]. As expected, designing and executing the scenarios exposed gaps in our understanding of the kernel behavior. Discussing the issues raised by the scenarios with ED's development team helped deepen our understanding of the required kernel behavior.

4.3 Application of a Mechanical Prover

TAME's specification and proof support significantly simplified the verification effort. Using TAME rather than hand proofs not only reduced the time required to complete the proofs, it also avoided some disadvantages of hand proofs, such as overlooking one or more cases.

4.4 Application of Refinement

For the approach in Section 3 to succeed, a security property must be preserved under refinement. It is well-known that safety (but not liveness) properties are preserved under refinement [1]. Hence, our techniques may be used to guarantee security properties that are safety properties. It is easy to show that all security properties verified for ED, except one, are safety properties and therefore

preserved by refinement. It is also easy to show that the one non-safety property, `No-Infiltration`, is also preserved under refinement.

5 Needed Research

5.1 Automatic Checking and Derivation of Code Annotations

For many years, researchers have recommended annotating code with precondition, postconditions, and invariants (see, e.g., [13]). Such code annotations are already used in practice: Developers at Praxis annotate SPARK programs with assertions and use tools to automatically check the assertions [4]. Moreover, in the largest Microsoft product groups, annotations are a mandated part of the software development process [6]. Unfortunately, manual annotation of source code remains rare in the wider software industry because it is highly labor-intensive [9]. To address this problem, Amtoft et al. have developed new techniques for checking *and* for deriving annotations from SPARK programs [2]. Tools for checking and deriving code annotation for programs written in other languages, such as C, would be extremely useful.

5.2 Automatic Generation of Tests from Assertions.

Many new techniques for constructing tests from formal specifications have been proposed. Such techniques (see, e.g., [7]) derive a set of test cases from a formal specification and use them to test the given program. One promising new approach constructs test cases from preconditions and postconditions and uses the test cases to check that the given program satisfies the asserted preconditions and postconditions. Validating source code in this manner should provide high assurance of both the security and functional correctness of source code. Reference [20] describes an approach that uses automatic test generation from preconditions and postconditions to find bugs in Java code. Similar research is needed that uses tests generated from an annotated program written in another language, such as C, to check the program for errors.

5.3 A Code Conformance Proof Assistant

The semantic distance between the abstract TLS required in a formal certification and a low-level program, such as a C program, is huge. While the TLS describes the security-relevant program behavior in terms of sets, functions, and relations, the description of the concrete program behavior is in terms of low-level constructs, such as arrays, integers, and bits stored in registers and memory areas. Hence, automatic demonstration of conformance of low level C code to a TLS may be unrealistic. A more realistic goal may be a proof assistant with two inputs, a C program annotated with assertions and a TLS of the security-relevant functions of that program. The assistant would help a user show conformance between the C program and the TLS.

5.4 Automatic Code Generation

One promising way to obtain high assurance that an implementation satisfies critical security properties is to generate code automatically from a specification that has been proven to satisfy the properties. Automatic code generation is already feasible for some low-level specification languages such as Esterel [18]. While constructing efficient source code from more abstract specifications is possible for simple program constructs using simple data types (see, e.g., [15]), new research is needed to produce efficient code from specifications containing richer constructs and data types. Such technology could drastically reduce the effort required to produce efficient code and provide high assurance that the code satisfies critical security properties.

6 Conclusions

One valuable byproduct of applying formal methods in software certification is that the process produces a formal specification of the required software behavior. Developing this specification has at least two benefits: 1) a formal specification can be invaluable when a new version of the software is developed, and 2) the process of developing a formal specification by itself may expose errors. Regarding the second point, the DO-178B certification standard for avionics software requires testing based on the Modified Condition Decision Coverage (MCDC) test criterion. However, to build MCDC tests, a formal specification must first be built. It has been observed that building this specification in itself uncovers software errors [17].

This paper has described one approach to applying formal methods in the certification of software. Building on existing software certification standards, such as DO-178B and the Common Criteria, more and improved approaches which use formal methods in software certification are needed. Applying these new approaches should have many benefits—the exposure of errors that, without formal methods, might not have been detected; software that has high assurance of satisfying critical security and safety properties; and the existence of a formal specification that would not have been built otherwise.

References

[1] Abadi, M. and L. Lamport, *The existence of refinement mappings*, Theoretical Computer Science **82** (1991), pp. 253–284.

[2] Amtoft, T., J. Hatcliff, E. Rodrguez, Robby, J. Hoag and D. Greve, *Specification and checking of software contracts for conditional information flow*, Technical report (2007).

[3] Archer, M., C. L. Heitmeyer and E. Riccobene, *Proving invariants of I/O automata with TAME*, Autom. Softw. Eng. **9** (2002), pp. 201–232.

[4] Barnes, J., "High Integrity Software: The SPARK Approach to Safety and Security," Addison-Wesley, 2003.

[5] *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 (2004).

[6] Das, M., *Formal specifications on industrial-strength code – From myth to reality*, in: *Proc., Computer-Aided Verification (CAV 2006), Seattle, WA*, 2006.

[7] Gargantini, A. and C. L. Heitmeyer, *Using model checking to generate tests from requirements specifications*, in: O. Nierstrasz and M. Lemoine, editors, *ESEC/FSE'99, 7th Eur. Software Engineering Conf. and 7th ACM SIGSOFT Symp. on Foundations of Software Engineering, Toulouse, France, Proceedings*, Lecture Notes in Computer Science **1687** (1999).

[8] Greve, D., M. Wilding and W. M. Vanfleet, *A separation kernel formal security policy*, in: *Fourth International Workshop on the ACL2 Prover and Its Applications (ACL2-2003)*, 2003.

[9] Hallem, S., B. Chelf, Y. Xie and D. R. Engler, *A system and language for building system-specific, static analyses*, in: *PLDI*, 2002, pp. 69–82.

[10] Heitmeyer, C., M. Archer, E. Leonard and J. McLean, *Applying formal methods to a certifiably secure software system*, IEEE Trans. Software Engineering **34** (2008), pp. 82–98.

[11] Heitmeyer, C. L., M. Archer, R. Bharadwaj and R. D. Jeffords, *Tools for constructing requirements specifications: The SCR toolset at the age of ten*, Comput. Syst. Sci. Eng. **20** (2005).

[12] Landwehr, C. E., C. L. Heitmeyer and J. D. McLean, *A security model for military message systems*, ACM Trans. Comput. Syst. **2** (1984), pp. 198–222.

[13] Meyer, B., *Applying 'design by contract'*, IEEE Computer **25** (1992), pp. 40–51.

[14] Owre, S., J. Rushby, N. Shankar and F. von Henke, *Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS*, IEEE Transactions on Software Engineering **21** (1995), pp. 107–125.

[15] Rothamel, T., C. Heitmeyer, E. Leonard and Y. A. Liu, *Generating optimized code from SCR specifications*, in: *Proc., ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES 2006), Ottawa, Canada*, 2006.

[16] Rushby, J., *Design and verification of secure systems*, in: *Proceedings, 8th ACM Symp. on Operating System Principles*, 1981.

[17] Rushby, J. (2006).

[18] SCADE Tool Suite. Tools and documentation available at http://www.esterel-technologies.com/products/scade-suite.

[19] Schneider, F. B., *Enforceable security policies*, ACM Trans. Inf. Syst. Secur. **3** (2000), pp. 30–50.

[20] Smaragdakis, Y. and C. Csallner, *Combining static and dynamic reasoning for bug detection*, in: *Tests and Proofs, First International Conference, TAP 2007*, Lecture Notes in Computer Science **4454** (2007), pp. 1–16.