A Formal Method Experience at Secure Computing Corporation

John Hoffman and Charlie Payne

Secure Computing Corporation
{john_hoffman,charlie_payne}@securecomputing.com

Abstract. We discuss the formal methods efforts for LOCK6, a secure operating system. We emphasize how the process of formal methods fit into the development process as a whole, and discuss the lessons learned from our experience.

1 Introduction

In this paper we discuss the formal methods efforts performed on the development effort of LOCK6, a secure operating system partially developed under contract to the U.S. Government. We describe from a high level the high assurance processes used, the motivations behind these processes, and the lessons we learned during the experience.

LOCK6 was an effort to create a new highly secure operating system, with a POSIX compliant interface. While the system itself has been deployed, the assurance component of the work was not completed because the original customer stopped funding for the project, and the new customer was not interested in funding the remaining assurance tasks. The reasons behind the original customer dropping funding were unrelated to the formal methods involved with the project.

The basic design of LOCK6 incorporated much of the operating system technology developed at Secure Computing over the past 10 years. Some of these technologies include the LOCK operating system [Say89], an operating system developed as a proof of concept that an operating system could be formally proved correct. Another collection of technologies were developed in the DTOS program, and some of its spinoff programs [Min95, FM93, Cor97, Fin96, CL98]. The primary goal of DTOS was to incorporate strong security mechanisms into the Mach micro kernel. Many of the architectural concepts and formal modeling techniques developed on DTOS were directly incorporated into the LOCK6 effort.

The primary goal of the LOCK6 effort was to create a B3+ operating system. B3 is defined in the Trusted Computing Security Evaluation Criteria [Cen85] (also known informally as the Orange Book). For a product to achieve B3 status, it has to have the following assurance evidence:

1. a complete formal model of the security policy
2. an informal model of the system

3. convincing arguments (informal proofs) that the model satisfies the security policy
4. evidence that the system model matches the system implementation

For LOCK6, the "+" in B3+ refers to additional additional assurance tasks that we performed. We created a formal model of the system suitable for performing a noninterference analysis. Noninterference is a characterization of a multi-level secure system. More accurately, it is a characterization of a model of a multi-level secure system. A model that satisfies noninterference is a model that contains no information flows from higher classification levels to lower classification levels. Noninterference was a large driver in the formal methods effort. Noninterference requires a detailed model because noninterference is not closed under refinement [McL94]. Consequently if anything is left out of the model, it could potentially be used as a means to illicitly downgrade information between levels. Thus, all error codes and all passed parameters need to be included in the model. We completely specified the operating system interface in the LOCK6 modeling efforts.

Developing an operating system to go through a B3 TCSEC evaluation requires a great deal of documentation. The five primary areas of assurance documentation developed for LOCK6 include a formal security policy model, an informal system model, a formal system model, informal proofs that the models satisfy the security policy, and evidence that the models correspond to the code. A significant effort was made to minimize the amount of documentation produced. A primary goal in our software development process is for the assurance evidence to be a byproduct of the development process. On previous projects at Secure Computing this was not the case. Assurance analysts would work largely independently of the designers, and have a parallel set of documentation. Later in the paper we will describe in more detail the five main areas of assurance documentation and how these documents fit into the development process.

The lessons learned from our experience are common and summarized below.

1. Formalizing natural language statements find mistakes and ambiguities.
2. Tightly linking the development and assurance processes is very beneficial.
3. Modeling is a significantly different skill from programming.
4. Table-based specifications are easy to grasp.
5. Good tools are important.
6. Good means of inter-team communication are necessary.

2 The System Architecture and Assurance Documents

2.1 Basic System Architecture

The basic architecture of LOCK6 has a Supervisor that provides interprocess communication (IPC), virtual memory, hardware interrupt, thread, and process management services. It is the only component to execute in privileged mode on the processor. Servers executing in unprivileged mode provide device, file

system, network, security, audit and logging services. Each of these servers has an interface that is built upon the IPC services provided by the Supervisor. One advantage of this approach is that with different servers operating in isolated address spaces, we are guaranteed that the servers can only interact through the IPC interface. No "back door" (accidental or intentional) manipulation of global data structures can occur. This removes one of the major concerns in secure system development.

2.2 Formal Security Policy Model

The formal security policy model describes at a high level what it means for the system to be secure. The basic policy was a multi-level secure (MLS) policy [Cen85] that included noninterference [FHOT89, Rus92, Fin90], with a mandatory non-hierarchical access control policy (aka Type Enforcement [BK85]), incorporated RBAC [SCFY96, Hof97] (Role based Access control) and also included Unix mode bits.

The policy is first specified in English as requirements that are incorporated into the requirements database tool used for the project. The security policy is essentially a requirements document for the system, and the database tool is used to track all requirements. Thus, the security policy requirements are all fed directly into the requirements documents for the system, and the security policy is written at the same time the system requirements documents are written.

Writing security policies is as difficult as writing any requirements document. In addition, for various programmatic reasons, the policy we wrote had many different authors. We found formalizing the policy a very useful exercise. Formalizing requirements helps to resolve the ambiguity inherent in English specifications. Our formalization effort caused numerous changes to the English when we realized how ambiguous it was and how many disagreements there were as to the meaning. Unambiguous English is of course impossible to write, but the process of formalizing the policy assisted in its clarification. For a point of reference, the English version of the policy was 100 pages long, and the English with the PVS formalizations interspersed was 150 pages long.

Placing requirements directly from the policy into the requirements process was first attempted at SCC in the LOCK6 effort. Most security requirements as written are untestable (e.g. No high level process shall be able to send data to a low level process unless it has special permission). Allocating this requirement through appropriate parts of the system raised visibility of these security requirements to developers. And the usual requirements traceability process made it easier to determine if the security requirements had been satisfied. This was a significant cultural shift for the testing staff, to accept requirements that may be untestable.

2.3 Informal Model

The informal model is an English statement of the interface of the operating system as seen by a client process. The model is a finite state machine, that

contains an abstraction of the system data structures. All parameters are specified for each system call. All possible return values are completely described in a stylized English and tabular format.

The model was created by developers with assistance from assurance personnel as part of the detailed design process of the system. The portion of the informal model that corresponded to each server was directly incorporated into the design documentation for the component, and was maintained by the appropriate developer. The model emulated the system design, in that each component was a stand-alone model. All 180 entry points into the operating system were modeled, with the informal model running roughly 500 pages long.

Putting developers in charge of the informal model greatly facilitated communication between the teams. One of the techniques we used to facilitate developer buy-in was to use a table-based approach. We found tables generally easier to read and understand than a more functionally oriented specification. An unfortunate consequence of our table based approach was that the tools we used to maintain the document were too cumbersome. LaTeX was used as the document production system, and the tables were too complicated to easily edit. In the future we intend to adopt better documentation strategies to facilitate this portion of the process.

Because we had developer buy-in, the developers owned the informal model. Developer ownership meant the model was maintained better than in the past, in part because it was used extensively by the developers. Inclusion of the model in the design documentation meant that while the document was larger than it may have been in the past, it reduced the redundancy between model and design, and consequently reduced the overall size of the documentation. This is one case where having a detailed model sufficient for noninterference greatly facilitated the process.

Developers found abstraction difficult to grasp, and defining detailed guidelines for how to abstract the model proved to be quite difficult. Having a model with significant abstraction would have made the model less useful for developers, who used the model as a reference during debugging. The informal model was the best description of the system outside of the code itself, and was a very effective means of communication between teams who needed to quickly understand the detailed operations of the entry points. Many developers found that the act of creating the model clarified the design. It forced them to think through the design completely, much as formalizing the policy helped clarify the security requirements.

2.4 Formal Model

The formal model is the informal model translated into a formal language.

Once the informal model was complete and agreed upon by developers and the assurance group, the developers began coding and the assurance group began formalizing the model. If ambiguity was found in the informal model, the design was checked and (if it existed) the code was reviewed. If the behavior of the system was appropriate, a bug report was filed against both the informal

and formal models to reflect the behavior of the system. If the system behavior was incorrect or undesirable, then a bug report was filed against the code. The entire model was written in PVS [Owr95, Owr93]. We made extensive use of subtyping within PVS. Typechecking the specifications found many of the common specification errors of incomplete bounds checking, and missed cases. However, few of these cases were missed in the code.

The basic framework of the formal model utilized SCC's composability framework [Cor97] began as part of the DTOS program, and was later extended as part of the Composability program. This framework borrows heavily from the Abadi-Lamport work [AL93] and allows the modeler to specify several different "components" (in our case, servers) that interact through an interface (in our case, IPC). Initially we had intended the supervisor to be modeled as another server. This created problems, because each server uses IPC to supply an interface to other servers and client processes. Thus, there were essentially two different layers of abstraction residing simultaneously in the model. This caused confusion among developers examining the model and among testers who needed to create test harnesses for the system. We began to address this problem using refinement techniques. We started to create a high level abstraction of IPC that could be invoked by the other servers in modeling their outcalls, and interfaces.

Much of the formal model was complete when the funding for the project was cut. It was quite a large model; on the order of 15K lines of specification. The large size was due to two significant factors: the size of the interface, and the level of detail in the model. One of the design goals of LOCK6 was to create a POSIX compliant interface. POSIX supports a rather robust collection of operations, and while we were able to push a significant portion of the POSIX processing into clients (who made use of libraries), the interface was still quite extensive. Since we wrote our specifications to support a noninterference analysis, all inputs, outputs, and error codes had to be specified. This made both the informal and formal models quite large, and potentially impossible to analyze. It should be noted however, that PVS is quite good at proving state invariants, and we believe that all of the invariant proofs necessary for a thorough analysis of the system would have been executed with a minimum of human interaction. This would have added a great deal of confidence to the otherwise informal proofs.

It is interesting to note how small a role the formal tools has played in this discussion. Good tools are crucial, and we found PVS to be a good tool if the user has sufficient sophistication. But much of the work associated with this development and formal methods effort went into aspects that were completely independent of the tool set. It is interesting to note also that experience with previous generations of verification environments made signing up for complete formal proofs appear much riskier than we now believe it would have been. PVS is a very robust system and has a number of features that cause us to believe that we would have been able to successfully formally prove many of the statements we had originally intended to prove informally. These features include the ability to create proof strategies, and the ability to automatically rerun proofs.

2.5 Formal Model Based Testing

Formal model-based testing is an orange book requirement for A1 systems (A1 systems are considered more secure than B3 systems). In this testing, the code is tested against a formal design level model to ensure the model and the system are consistent. Formal model-based testing checks that all required behaviors are exhibited, and that no undesirable behaviors occur.

Due to the cut in funding we were unable to complete the formal model based testing. However, we did create a process and tested it on some of the smaller components of the system. Test cases were developed from the tables in the informal model. Each test case was translated into a PVS theorem, and proven. Each test case was also translated into code, and a test harness was created to dump the server data structures before and after the test ran. These data structures were then compared to ensure only the appropriate portions changed.

We found the model to code correspondence to be difficult to establish. Although our model was detailed, the code had far more detail than the model. Separating the PVS analysis from the code testing allows the two activities to proceed in parallel. This was very useful since the implementation was changing far more rapidly than the model. The informal model proved to be more readily understandable to the testers than the formal model, and it was easier to create the test cases from. We also found that specific proof strategies were needed to allow us to repeatedly rerun the proofs. PVS has many high level powerful proof commands, but often ran quite slow on our specifications. Some modest customization of these proof commands created significant improvements in performance.

3 Conclusions

Some of the lessons learned by our recent (albeit incomplete) effort in formal methods showed us that developer buy in and ownership is crucial. The LOCK6 assurance effort was on track to complete under budget once the funding was cut. A critical reason was that developers owned the informal model and used it. Thus it was in their interest to keep it maintained and current. The bug tracking and fixing process was set up such that changes to the informal model spawned change requests to the formal model. It was therefore possible to always know the delta between the informal and formal models. This is something that in previous programs had been quite difficult.

Useful incorporation of formal methods into software development requires a very mature development process. Extensive and detailed communication must occur between many different teams, and these communication channels must be very clearly defined and rigorously followed in order to avoid extensive rework and confusion. This is perhaps not too surprising, since it has been long been argued in the community that formal methods only make sense in a mature environment.

We did not work hard enough to add additional abstraction to the formal model. The formal model was at too low a level of detail to make it analyzable.

However, having the detail in the informal model was important for developers and testers. They could use the document to understand the exact logical conditions that caused their error returns. If we had abstracted these out of the informal model, the developers would not have found it as useful. What was needed was a clear criteria for how to abstract error messages in the formal model without abstracting out potential covert channels. Moreover, it is imperative that the mapping between the models be clear and easy to follow. Often different people are required to maintain the specifications, and since a typical transition could take two or three pages to completely describe, the specification needed to match the informal model in an obvious way to facilitate comprehension and maintenance.

Although little testing was performed to verify the correctness of the formal model, we had developed tooling to perform the testing. The basic process was that test cases were developed from the informal model and coded into the system. From each test case a PVS theorem was defined, and then proved. The proof of most test cases was very similar. It was our intention to develop PVS proof strategies that would prove the majority of these theorems.

As with code, specification conventions are important. It makes reading specifications by different authors easier to read, and increases the flexibility of staffing and loading.

In summary, even though some of our processes are 10 years old, there always seems to be significant opportunities for improvement.

References

[AL93] Martin Abadi and Lesli Lamport. Conjoining specifications. Technical Report 118, Digital Equipment Corporation, Systems Research Center, December 1993.

[BK85] W.E. Boebert and R.Y. Kain. "A Practical Alternative to Hierarchical Integrity Policies". In *Proceedings of the 8th National Computer Security Conference*, pages 18–27, October 1985.

[Cen85] National Computer Security Center. Department of Defense Trusted Computer System Evaluation Criteria. Technical report, US National Computer Security Center, NCSC, Fort Meade, Maryland, 1985.

[CL98] Michael Carney and Brian Loe. A comparison of methods for implementing adaptive security poli cies. In *Seventh USENIX Security Symposium Proceedings*, pages 1–14, San Antonio, TX, January 1998. USENIX Association.

[Cor97] Secure Computing Corporation. DTOS Composability Study. Technical report, 1997. http://www.securecomputing.com/randt/HTML/technical-docs.html.

[FHOT89] Todd Fine, Thomas Haigh, Richard O'Brien, and Dana Toups. Noninterference and Unwinding for LOCK. In *Proceedings of Computer Security Foundations Workshop II*, pages 22–28, Franconia, NH, Jun 1989. IEEE.

[Fin90] Todd Fine. Constructively Using Noninterference to Analyze Systems. In *IEEE Symposium on Security and Privacy*, pages 162–169, Oakland, CA, May 1990.

[Fin96] Todd Fine. A framework for composition. In *Proceedings of the Eleventh Annual Conference on Computer Assurance*, pages 199–212, June 1996.

[FM93] Todd Fine and Spencer E. Minear. Assuring Distributed Trusted Mach. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 206–218, May 1993.

[Hof97] John Hoffman. Implementing RBAC on a Type Enforced System. In *Proceedings of the Thirteenth Annual Computer Security Applications Conference*, pages 158–163, 1997.

[McL94] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1994.

[Min95] Spencer E. Minear. Providing policy control over object operations in a mach based system. In *Fifth USENIX Security Symposium Proceedings*, pages 141–156, Salt Lake City, UT, June 1995. USENIX Association.

[Owr93] Owre, Shankar and Rushby. The PVS Specification Language (Beta Release). User Manual, SRI International Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, June 1993. http://www.csl.sri.com/reports/pvs-language.dvi,ps.Z.

[Owr95] Owre, Shankar, Rushby, Crow and Srivas. A Tutorial Introduction to PVS. User Manual, SRI International Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, June 1995. http://www.csl.sri.com/sri-csl-fm.html.

[Rus92] John Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, December 1992. http://www.csl.sri.com/csl-92-2.html.

[Say89] Sami Saydjari. LOCK Trek: Navigating Uncharted Space. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1989.

[SCFY96] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.