Cost Profile of a Highly Assured, Secure Operating System

Richard E. Smith
Secure Computing Corporation

rick_smith@securecomputing.com

March 19, 2001

Abstract

The Logical Coprocessing Kernel (LOCK) began as a research project to stretch the state of the art in secure computing by trying to meet or even exceed the "A1" requirements of the Trusted Computer System Evaluation Criteria (TCSEC). Over the span of seven years, the project was transformed into an effort to develop and deploy a product: the Standard Mail Guard (SMG). Since the project took place under a U. S. government contract, the development team needed to maintain detailed records of the time spent on the project. The records from 1987 to 1992 have been combined with information about software code size and error detection. This information has been used to examine the practical impacts of high assurance techniques on a large scale software development program. Tasks associated with the A1 formal assurance requirements added approximately 58% to the development cost of security critical software. In exchange for these costs, the formal assurance tasks (formal specifications, proofs, and specification to code correspondence) uncovered 68% of the security flaws detected in LOCK's critical security mechanisms. However, a study of flaw detection during the SMG program found that only 14% of all flaws detected were of the type that could be detected using formal assurance, and that the work of the formal assurance team only accounted for 19% of all flaws detected. While formal assurance is clearly effective at detecting flaws, its practicality hinges on the degree to which the formally modeled system properties represent all of a system's essential properties.

1. Introduction: Multi-level Security and Trusted Computing

Starting in the 1960s, members of the military and intelligence communities sought a computing system that could safely share information with different degrees of confidentiality [1,2]. Systems achieving that goal were said to operate in *multi-level* mode, which indicated that the system contained information at several different levels of confidentiality but did not share them with unauthorized users [3]. Communities requiring multi-level systems also demanded a high level of confidence that they operated correctly, since leaked intelligence data could potentially destroy the effectiveness of extremely expensive intelligence gathering systems. Furthermore, there were serious legal penalties if a person set up a system that leaked data: the author has heard defense officials informally refer to multi-level computers as "felony boxes."

Following a number of experiments, developers focused on a lattice-based access control mechanism that individual users could not disable or bypass [4,5,6]. Such systems could process data items that carried different security classifications, and allowed users to share the data safely without "leaking" sensitive data to unauthorized users. The Multics system hosted an early, successful implementation of this mechanism, based on a model of hierarchical access control developed by Bell and LaPadula [7].

In this model, the system assigned security classification labels to processes it ran and to files and other system-level objects, particularly those containing data. A process was always allowed to read and write an object if the process' label matched the object's label (subject to other permissions, like those linked to file ownership). A process was also allowed to read the data in an object if the object's classification label was lower than the process' label. For example, a "secret" process could read data from an "unclassified" object, but never vice versa. However, the mechanism explicitly forbade processes from writing data to an object when the object's classification label was lower than the process' label. This prevented programs from acci-

dentally leaking classified information into files where unauthorized users might see the data. More importantly, it prevented users from being tricked into leaking information by "trojan horse" programs. The mechanism was also called *mandatory access control* because it was always enforced and users could not disable or bypass it.

Although this basic multi-level security mechanism seemed simple to implement, it proved very difficult to implement effectively. Several systems were developed that enforced the security rules, but experimenters quickly found ways to bypass the mechanism and leak data from high classification levels to low ones. The techniques were often called *covert channels* [8]. These channels used operating system resources to transmit data between higher classified processes and lower classified processes. For example, the higher process might systematically modify a file name or other file system resource (like the free space count for the disk or for main memory) that is visible to the lower process in order to transmit data.

Multi-level security may have posed a serious challenge to operating system developers, but it posed a more difficult challenge for customers who needed such systems. While it was simple enough to tell if a system provided multi-level security mechanisms, there was no way to tell if the system also had covert channels or other security problems. To address this problem, the U.S. government developed a process for evaluating the security of operating systems and published a set of standards: the Trusted Computer Systems Evaluation Criteria (TCSEC) [9,10]. Operating system vendors could submit products for evaluation against the criteria.

The TCSEC established a scale for defining operating system security that consisted of letters (A, B, C) and numbers (1, 2, 3) to define different levels of security functionality and different levels of confidence in the security's effectiveness. The TCSEC defined the requirements to achieve the following levels in increasing order of security: C1, C2, B1, B2, B3, A1. The B and A level systems were required to provide multi-level security.

The TCSEC referred to the security critical elements of a computer system as the *trusted computing base* (TCB) and focused the evaluation effort on those elements. The A1 requirements incorporated all of the lower level requirements that were believed to be most effective, and represented the highest level of operating system security that seemed practical at that time. To meet high assurance requirements, a TCB was supposed to implement the *reference monitor* concept [4], which contained three essential features:

1. Nonbypassable: the TCB is always invoked when the system performs an operation that is covered by the system's security policy, and it applies the established access control policy when it is invoked. On an A1 TCB, this must happen every time a software process (a "subject") attempts to access a distinct data item (an "object). There must be no way to bypass the TCB when the system performs such an operation.
2. Tamperproof: the TCB itself and the resources that control its operation must be protected from modifications that could affect how it enforces security. An A1 TCB also needs explicit mechanisms to protect and verify the integrity of the TCB.
3. Assurance: the TCB must be small and simple enough so that independent evaluators can examine it and determine with confidence that the TCB correctly enforces security. To meet A1 requirements, the TCB needs a formal specification of its security objectives, a formal description of its design, and copious evidence that the TCB implementation complies with both the design specifications and the security requirements. In any case, the TCB software has to be small and compact in order to make that analysis feasible.

Unfortunately, the TCSEC and its associated evaluation program failed to win significant support among commercial vendors [11]. Evaluations proved to be incredibly costly, and most vendors found it impractical to provide multi-level security features in their standard products. At best, vendors would submit their off-the-shelf products for a C2 evaluation.

Operating systems with multi-level security generally appeared as separate products, with higher costs and lower performance than the same vendor's standard operating systems. Moreover, the vast majority of multi-level security vendors provided products for B1 evaluation, even though the Defense Department's guidance for secure systems deployment did not recommend B1 systems for multi-level applications [12].

Only a handful of vendors took up the challenge and tried to produce an A1 product. Examples include Honeywell's SCOMP [13], Digital Equipment Corporation's VAX security kernel [14], Gemini's GEMSOS [15], and the Boeing MLS LAN [16]. Only SCOMP, GEMSOS, and the MLS LAN products completed A1 evaluations. These systems tended to be costly, special purpose systems that did not support the programming interfaces of more familiar operating systems. None of these systems attracted a market outside of a very narrow, defense-oriented community. Although Digital's security kernel never achieved an A1 evaluation, the kernel found its way into at least one commercial product, the VAX 8800 model [17]. This was the only case of an A1 development program begetting a true commercial product, until the LOCK program.

2. The LOCK Program

The LOCK program was started by the U. S. National Computer Security Center to stretch the state of the art in secure computing system development [18,19]. LOCK built upon earlier government research efforts, notably the Provably Secure Operating System [20] and the Secure Ada Target (SAT) work [21]. The centerpiece of the LOCK effort was the development of a general purpose TCB that met or exceeded the TCSEC's stringent A1 requirements. Other objectives included the integration of cryptographic services into a TCB and demonstrated performance comparable to similar unsecured machines.

In 1987, Honeywell Corporation was awarded the contract for the LOCK program, and assigned it to the team that had performed the SAT work. In 1989, Honeywell divested itself of computer projects and 'spun off' the LOCK team into a private corporation, which became Secure Computing Corporation. Table 1 presents the project's chronology. The development effort detailed in this paper started with the LOCK program in 1987 and ended in 1992, costing approximately $23 million. Additional statistics from follow-on activities are also included to place the reliability of the LOCK prototype in context. The statistics and other information presented in this paper is constrained in two ways. First, the paper is limited to presenting information that has been explicitly released for publication by the government and, in a few cases, more detailed information may exist in unreleased project archives. Second, the paper is limited to whatever information was collected during the project, and it isn't possible to go back and collect different information today.

By 1994 a significantly improved version of the LOCK system was developed and deployed as the Standard Mail Guard (the SMG) [22]. Those enhancements cost approximately $9 million over a 2 year period. The Guard is currently used to provide multi-level electronic mail connectivity in a variety of military and government sites. In addition, its underlying security architecture formed the foundation for the Sidewinder, a commercial Internet firewall [23,24].

Date	Milestone	Key Activities
1987	LOCK program started	emphasis on formal assurance
1990	Initial contract completed	internal R&D, product work started
1991	LOCK follow-on contract executed	networking added under internal R&D
1992	Demonstration program executed	LOCK size and cost statistics collected
1993	SMG work started	emphasis on reliability through testing
1994	First SMG installed and operational	passed 30 day stress tests

Table 1: LOCK and SMG Project Chronology

The LOCK system was designed and built in the form of an interactive multi-user system that connected to users via character based ASCII terminals. This was the prevailing form of medium and large scale systems at the time, and reflected many of the requirements and implicit design assumptions of the TCSEC. This form also appeared to support the broadest range of potential uses in addition to time-shared interactive access. One could scale it down and run it on a workstation driven by a modern microprocessor, or perhaps scale it up and run batch jobs under a job monitor. However, character oriented graphics were the prevailing technology at that time, so LOCK did not include any workstation graphics or windowing capabilities. The TCSEC was silent on the subject of network access – its Trusted Network Interpretation was published in 1987 as the LOCK program began [25] – and LOCK was based on the TCSEC. The LOCK designers assumed that networking could be added at a later date. In fact, TCP/IP support was added to LOCK using corporate R&D funds in 1992.

LOCK incorporated a number of features in an attempt to produce a practical system that complied with the A1 requirements:

- TCB separated from the operating system

  To keep the TCB as small as possible, the LOCK TCB only contained the mechanisms that were essential for security critical activities. This included a "storage manager" that provided a memory mapped, flat file system with random, system-assigned identifiers instead of file names. The TCB also provided enough process management to support multi-user operation.

  Other "operating system" functions were provided by LOCK/ix, a customized, POSIX compliant version of Unix that served as the platform for implementing user applications [26]. LOCK/ix provided a Unix file system for applications, hosted atop the LOCK storage management primitives. LOCK/ix ran in user mode and all of its operations, including Unix "superuser" functions, were subordinated to LOCK's security enforcement mechanisms. Each user ran a separate instance of LOCK/ix, and separate instances ran at different security levels if a user had Unix processes running at different security levels.

  The resulting TCB contained 115,887 lines of code (LOC), written almost entirely in the high level language C.

- Minimized kernel mode software

  Less than 1,000 lines of C source code ran in the processor's kernel mode on LOCK. Most TCB functions ran in user mode, including the storage manager, process managers, and all device drivers. This provided increased confidence in LOCK's security.

- Military grade cryptography integrated with the TCB

  Prior to LOCK, government approved encryption was generally implemented as a "black box" external to the computer system. Under the LOCK program, the encryption system was integrated into the TCB. Although its primary application was to encrypt mass storage, it also provided other services like random number generation and integrity sealing. LOCK's encryption system had to meet government cryptographic endorsement requirements in addition to TCSEC requirements.

- Hardware coprocessor for security decision making

  LOCK incorporated a component called the System Independent Domain Enforcing Assured Reference Monitor (SIDEARM) to handle all security decision making and event auditing. The SIDEARM was a separate, stand alone system connected to the LOCK host via a high speed shared memory. When a process on the LOCK host tried to open a file or perform some other object access, it would contact the SIDEARM to make the access control decision. The SIDEARM maintained the authoritative copies of all access control information. For most TCB operations the SIDEARM would automatically generate audit records as required by the TCSEC.

  The hardware separation provided two advantages. First, it reduced the risk of malicious users accessing and corrupting the security decisionmaking logic and databases. Second, it provided a

measure of parallelism since other processes could continue to run while a process awaited the results of an access control decision.

- Performance somewhat comparable to a standard Unix system

  The government established the goal that LOCK achieve at least 80% of the performance of a comparable Unix system. LOCK was implemented on the Motorola MC68020 microprocessor during the golden age of MC680x0 based Unix vendors. A small vendor, who we believe is no longer in this business, was selected as the source for LOCK hardware. That vendor's own version of Unix was used to benchmark LOCK's performance.

  Performance benchmark requirements were informally established with the government. Following a series of performance enhancements, LOCK (with its SIDEARM coprocessor) was measured as performing at approximately 97% of the vendor's off-the-shelf single processor MC68020 system. Measurements were based on the "AIM Suite III Multiuser Benchmark," a commercial Unix benchmarking package from AIM Technologies. When chosen early in the LOCK program, AIM claimed to offer "industry standards for evaluating Unix systems performance," and provided a list of over 150 customers [27,28].

- Practical use of the *type enforcement* mechanism in security software

  In addition to the multi-level security mechanism required of all B- and A-level evaluated systems, the TCSEC also required a mechanism to protect system integrity. LOCK introduced the *type enforcement* mechanism to do this. The mechanism unconditionally restricts data sharing between different sets of processes called domains. Data items with different access restrictions are assigned to different types. A table defines what operations a domain may perform on data of different types [29]. Unlike the discretionary access controls in typical commercial operating systems, the type enforcement mechanism is always applied and can not be bypassed or disabled by privileged processes or users.

  The type enforcement mechanism provides a way to assemble a provably nonbypassable security facility from a collection of separate software components, also called an *assured pipeline* [30]. The mechanism also represent key features of Clark-Wilson integrity policies [31]. Within the LOCK system, the type enforcement mechanism provided a way to reliably add new TCB software to LOCK without adding to its basic kernel. Type enforcement permissions granted these kernel extensions special access to security critical data to perform security critical operations.

3. LOCK Program Activities

The LOCK program incorporated hardware and software efforts into a waterfall-style development program. Figure 1 portrays the flow of major software development activities. The LOCK requirements specification process began with a detailed analysis of the TCSEC A1 requirements. This yielded a document called the "Annotated TCSEC" that interpreted the requirements in a precise and consistent fashion [32]. A large part of this consisted of precise definitions of terms and reconciliations of these definitions with the published words of the TCSEC. This process uncovered a number of ambiguities that could have produced security flaws or other pitfalls in the implementation. The Annotated TCSEC essentially captured the rationale by which the LOCK project complied with TCSEC requirements, so the document was ultimately transformed into the Philosophy of Protection document required in TCSEC evaluations.

The LOCK technical specification documents followed the sequence of A level specification, B level specification, and C level specification, as defined in the appendices of MIL-STD-490A [33], published by the U. S. Department of Defense (DOD). This yielded a specification structured in a three level hierarchy. Top level system specifications identified by the Annotated TCSEC were captured in the "A-spec" document. The B-specs captured subsystem design information, while the C-specs captured software component designs. The B-spec and C-spec documents were produced during the subsequent design phase. In addition, the LOCK team produced an Interface Control Document for each LOCK subsystem.

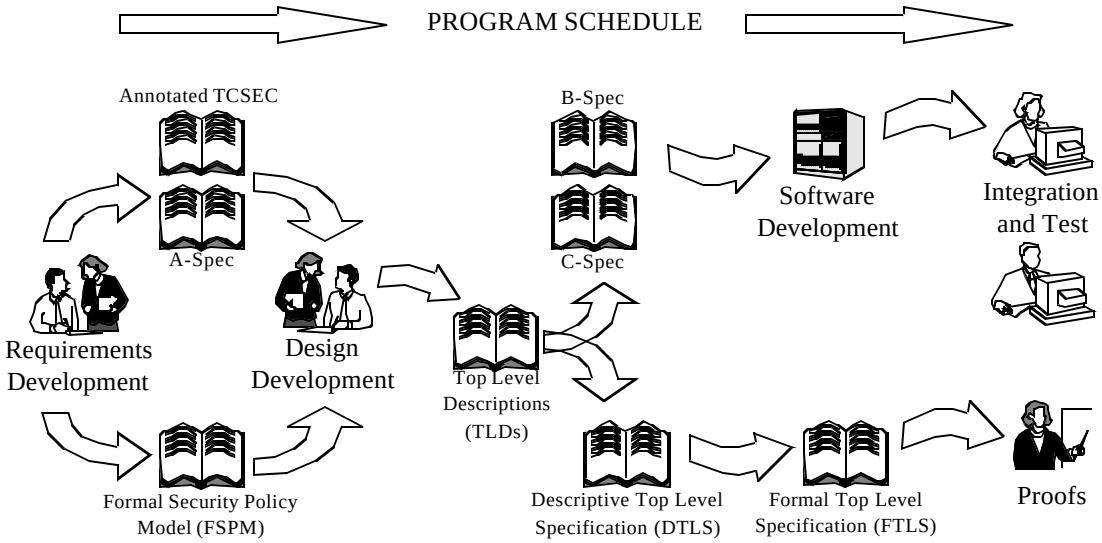

Figure 1: Software development took place in parallel with assurance activities

The software development team collaborated with the assurance team during LOCK's early phases. Once high level specifications were complete, the developers focused on implementation (top of diagram) while the assurance team focused on completing the formal specifications and proofs (bottom of diagram). The proof results were finished after the software entered its testing phase. A similar schedule applied to individual releases of the SMG.

Table 2 summarizes the LOCK software development costs, based on detailed costs presented in Section 4. The costs of the system engineering process encompass system wide development of high level requirements, and associated management and documentation activities. Costs for the software categories all include the conventional software development tasks: detailed design, coding, and unit testing. The TCB formal assurance category reflect the extra tasks required to meet the TCSEC A1 formal assurance requirements, as illustrated along the bottom of Figure 1. This includes the formalization of security requirements. The media encryptor software was subject to government cryptographic endorsement requirements, which mandated particular assurance tasks. Unlike the analytical activities applied to the TCB, these tasks focused on detailed requirements analysis and design rationales, not on formal methods. The integration and test category includes the work of integrating components, locating the bugs that keep them from coming together, and performing large scale tests of the result. Table 2 does not reflect program management or hardware costs.

Activity	Labor Hours	%
System Engineering	28,848	14%
TCB Software	64,448	30%
TCB Formal Assurance (Specifications and Lemmas)	37,581	18%
LOCK/ix Software	18,250	9%
Encryptor Software	19,490	9%
Encryptor Cryptographic Assurance	6,582	3%
Integration and Test	36,192	17%
Total	211,391	100%

Table 2: Cost of LOCK Software Development Activities

Table 3 rearranges the data in Table 2 to help illustrate the cost impact of high assurance on the LOCK program. The columns “excluding” high assurance tasks simply duplicate the corresponding entries from Table 2, and estimates what LOCK may have cost if high assurance activities had not been performed. The columns “including” high assurance have incorporated the high assurance activities into the corresponding software development activities: the TCB assurance costs have been added to TCB software development, and cryptographic assurance costs have been added to the media encryptor. To simplify the assurance work, LOCK/ix software was not part of the TCB, and so it required no additional high assurance tasks.

The rearranged data shows that high assurance tasks added 26% to the overall LOCK program costs. If we treat formal assurance as an increment over existing TCB software development costs, the added tasks yielded a 58% cost increase. In the same vein, cryptographic assurance added 34% to the media encryptor development cost. These costs only reflect the labor hours incurred by specifically identified high assurance activities. The figures do not capture either cost increases or cost savings that high assurance may bring about in other, conventional activities. The figures also fail to take into account system level activities associated specifically with the TCB or with the media encryptor. For example, the 58% figure would undoubtedly be reduced if we could accurately include other system level costs as well as back-end test and integration costs. Unfortunately, it is not clear how to assess these costs more accurately with the available LOCK data.

Activity	Omitting High Assurance Costs		Including High Assurance Costs		Percent Cost Increase for High Assurance
	Labor Hours	%	Labor Hours	%	
System Engineering	28,848	17%	28,848	14%	
TCB Software	64,448	38%	102,029	48%	58%
LOCK/ix Software	18,250	11%	18,250	9%	
Encryptor Software	19,490	12%	26,072	12%	34%
Integration and Test	36,192	22%	36,192	17%	
Total	167,228	100%	211,391	100%	26%

Table 3: Impact of High Assurance Activities on Program Costs

3.1. LOCK High Assurance

High assurance on the LOCK program was pursued in two dimensions. The LOCK TCB applied formal assurance methods to the TCB design. The media encryptor applied the rigorous procedures required to earn a U. S. government cryptographic endorsement.

Formal LOCK TCB assurance activities began with two major activities: policy specification and design specification. The specifications were written in Gypsy, a specialized language for formal specifications [34,35]. The formal policy specification was captured in the Formal Security Policy Model (FSPM) document. The FSPM described the critical security requirements of LOCK in a formal, mathematical model. In accordance with the TCSEC, these requirements centered around multi-level security. The LOCK FSPM used the concept of noninterference to model the multi-level security objective [36].

LOCK’s formal design specification was captured in its Formal Top Level Specification (FTLS) document. The FTLS contained a formal specification of the TCB’s interface that rigorously defined the behavior of every facility the TCB provided. The FTLS was developed in conjunction with the software design process that produced B-specs and C-specs for LOCK software subsystems. Originally, there were two separate design representations, a formal specification used for verification and a module specification used to guide the implementation. However, developers and verifiers found it difficult to work together with different repre-

sentations. To solve this, the LOCK program developed a top level description format that shared the essential information with developers and verifiers. The top level description (usually called a "TLD") was written in terms recognized by the developers and included the level of operating details (states, outputs, error conditions, etc.) required by the mathematicians responsible for formal verification. The top level descriptions were published in the Descriptive Top Level Specification and, during the SMG program, also appeared in software design documents.

Unlike the FTLS and FSPM, top level descriptions were not written in Gypsy. Instead, each description was formatted so it was relatively simple to generate a Gypsy specification based on it. When a top level description was changed it was relatively easy to update the corresponding Gypsy specification to reflect the change. To prove consistency between the formal requirements of the FSPM and the design as stated in the FTLS, the specifications were processed with the Gypsy Verification Environment (GVE).

Formal verification on the LOCK program focused entirely on validating the designed behavior of the TCB interface. Verification was not applied to the source code itself. Early work on the LOCK program attempted to represent the lowest level code specifications in Gypsy (the C-spec) but the specifications became too complex for the late-1980s Gypsy software to handle.

The media encryption software was developed in accordance with government requirements for military grade cryptographic systems. These requirements were different from those applied to an A1 TCSEC system, but they were intended to achieve a similar purpose: to reduce the risk of security flaws in the device being built. The cryptographic requirements were much more specific than the TCSEC requirements: they defined specific technical properties that were expected of a highly secure cryptographic device. The device's design had to be documented in its Theory of Equipment Operation document. A companion document, the Theory of Compliance, described the rationale by which the cryptographic security requirements were fulfilled by the device's design. This document served a similar purpose to the Philosophy of Protection required in TCSEC evaluations.

The purpose of the LOCK TCB's formal assurance work was to establish high confidence that the system achieved essential security objectives. Figure 2 illustrates how the A1 assurance activities of the TCSEC were intended to validate LOCK's functional and security requirements. The left-hand side illustrates the for-

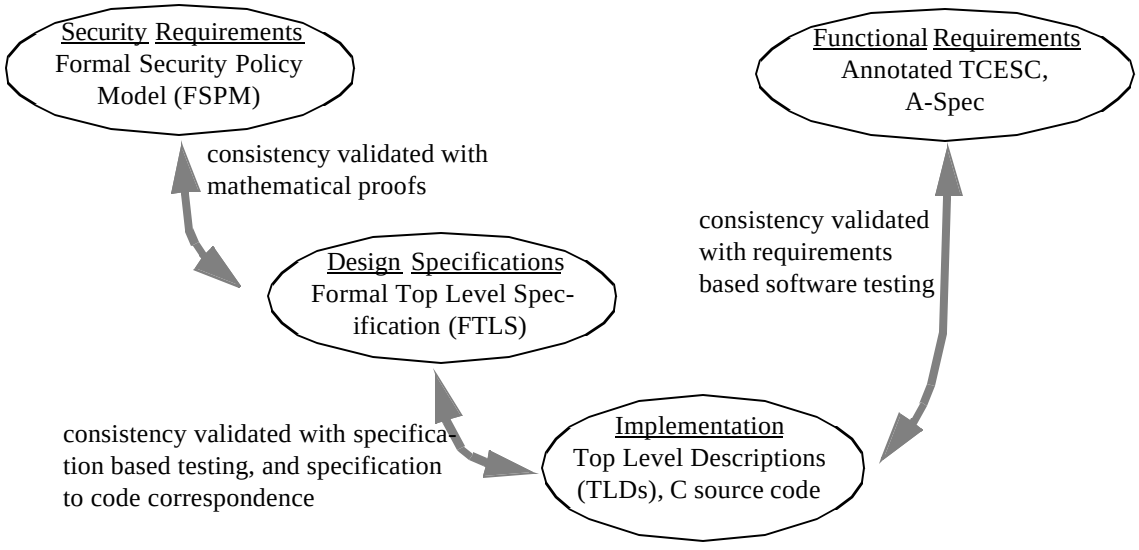

Figure 2: Formal assurance shows there are no inconsistencies between system elements

The formal assurance development process used on LOCK validated consistency between requirements, design, and implementation. Consistency between security requirements and system design is proven using mathematical logic. Inconsistencies indicate potential security flaws that, if not corrected, could produce flaws in the implementation.

mal verification activities while the right hand side represents validation activities more commonly associated with software development projects. All of these activities support an essential goal: to identify flaws and inconsistencies that might reflect security weaknesses in the LOCK system implementation.

However, there is a crucial distinction between formal activities on the left hand side and the engineering oriented activities on the right hand side. The formal activities only affected security objectives that were explicitly represented in the formal security policy and the formal design specifications. The engineering activities applied to all documented system requirements and objectives. There was no requirement to formally represent performance or availability requirements, so the formal methods were unable to identify design problems that might affect such requirements. On the other hand, test activities on the right hand side could at least demonstrate compliance or uncover non-compliance with most system requirements. Neither strategy was 100% successful at detecting system flaws.

3.2. LOCK Security Evaluation

The LOCK program ended without completing a formal A1 evaluation. A combination of cost and schedule constraints made the A1 evaluation impractical. Instead, LOCK and the subsequent SMG produced a detailed set of specification and analysis documents, and the documents themselves were used in lieu of the A1 evaluation to provide evidence of its security assurance. Prior to deployment, the SMG was subjected to the DOD mandated procedures for the certification and accreditation of computers handling classified information [3,37]. The SMG accreditation decisions by individual sites and by the Defense Information Systems Agency were based on the LOCK assurance documents combined with extensive third-party certification testing.

Here is a summary of major A1 assurance requirements with an indication of the degree to which each was completed on the LOCK program.

- Formal specification of the security policy and top level design - YES

  This was a major element of the LOCK program. The formal specifications were completed and kept up to date during and after the LOCK program. The practice continued on the SMG program. The specifications were independently reviewed by government selected consultants under the LOCK and SMG programs.

- Formal proof that the design is consistent with the policy - YES

  This was a major element of the LOCK program. The formal proofs were completed and kept up to date during and after the LOCK program. The practice continued on the SMG program. The proofs were independently reviewed by government selected consultants under the LOCK program.

- Formal analysis for covert channels that violate the multi-level security policy - YES

  This was a major element of the LOCK program, which made extensive use of noninterference to explicitly prove the limited extent of LOCK's covert storage channels. Noninterference played a smaller role in SMG because of changes in the security objectives. The proofs were independently reviewed by government selected consultants under the LOCK program and under SMG.

  The LOCK program also performed source code level analysis of storage and timing channels on a limited, experimental basis.

- Show correspondence of the formal specifications to the source code - PARTIAL

  The LOCK program addressed this problem with two different strategies, neither of which was applied to the entire TCB. The first was a "specification to code correspondence" effort that systematically mapped the correspondence between the FTLS and the LOCK TCB source code. This activity was perform in addition to conventional source code review that occurred as part of the software development process. The effort was completed for LOCK data structures and for two TCB services. The second strategy was to perform "specification based testing" that compared the

results of tests run on the LOCK TCB against the results of the same tests running in a simulation driven by the FTLS [38]. These tests were performed on a single TCB service to demonstrate the feasibility of the concept. These activities were independently reviewed by government representatives and consultants under the LOCK program.

- Show compliance with system architecture requirements - YES

  The TCSEC specified that all high assurance TCBs should comply with certain software design requirements that appeared to promote stable and maintainable software. These included protection for TCB components, process isolation, modularity, least privilege, conceptual simplicity, layering, abstraction, data hiding, and minimized complexity. The LOCK Philosophy of Protection described how the architecture and the project's software development process ensured compliance with these requirements. Insofar as these features were reflected in the system design or in system level behavior, they were independently reviewed by government representatives and consultants under the LOCK program.

- Configuration management - YES

  All LOCK documents and code modules were stored in a source code control system from the inception of the project, and the practices remained in force during all subsequent LOCK related activities, including the SMG program. The project used a highly customized configuration of the Source Code Control System (SCCS) of AT&T Unix [39]. The repositories, procedures, and compliance with them were periodically reviewed by government representatives and by ISO 9001 auditors.

- Security testing - YES

  Security testing on LOCK consisted of component level testing that exercised weaknesses previously reported in other operating systems. There was also a limited amount of specification based testing described earlier. LOCK was never subjected to independent security testing. During the SMG program, security testing was significantly expanded. Independent security testing of SMG took place as part of certification and accreditation activities.

In addition to the shortfall in specification to code correspondence noted above, LOCK fell short of A1 on the basis of documentation and independent evaluator review. Although documents were written to comply with all of the A1 requirements, and those documents were reviewed by government representatives, that did not guarantee an inexpensive, straightforward evaluation. The specific evaluators assigned to a LOCK evaluation might very well have required costly revisions of the existing documents. Likewise, the government's oversight of LOCK design and development did not unconditionally guarantee that a LOCK evaluation team would have endorsed all of LOCK's architecture and design decisions when studied in detail. As a new and struggling company, Secure Computing could not afford to risk the resources that the complete evaluation process might ultimately require. Instead, Secure Computing provided the assurance documentation directly to customers, who in turn used the documentation instead of a formal evaluation to support certification and accreditation decisions.

4. LOCK Productivity Statistics

Table 4, which appears later in this section, presents the total labor hours and lines of code comprising the LOCK system, as measured in June of 1992. The table was produced by sorting the labor hour reports provided by project personnel into categories associated with software coding, assurance tasks, and other major project activities. At the same time, the source code file sizes were measured and sorted into comparable categories.

4.1. Labor Hours

Since the LOCK program was a government contract, all labor hour data was collected in compliance with legal requirements prescribed by the Department of Defense Federal Acquisition Regulation Summary (DFARS) [40]. Organizations that perform government contracts were required to have a system in place to accurately collect cost information; the system had to be approved by the government to fulfill the DFARS requirements. A typical system consists of a Work Breakdown Structure (WBS) and a set of procedures to collect labor hours from employees. The WBS individually identified all of the tasks being performed under the contract: the LOCK program contained over 318 separate tasks. The collection procedures were established for the LOCK program and the procedures were systematically audited to ensure that labor hours were collected in a timely and reliable manner.

The LOCK program always operated under such a system, first within Honeywell, and then at Secure Computing. When Honeywell 'spun off' Secure Computing, the cost history of the LOCK program was transferred to Secure Computing's newly established costing system, so that no labor cost information was lost. Reviews and audits by the Defense Contract Audit Agency ensured that both organizations did an adequate job of tracking program costs. Moreover, both organizations were obligated under the DFARS to account for internal R&D costs using the same system, so LOCK's internal R&D labor costs are likewise accounted for accurately. Although not all of the labor hours in Table 4 are relevant to discussions within this paper, all collected labor costs are included here for completeness. Note that these labor hours don't represent the entire cost of the LOCK program: costs for some equipment, travel, and overhead expenses do not appear here.

4.2. Source Code Size

LOCK source code was produced in a variety of languages, but most LOCK software was written in C. The few cases that required assembly language were embedded in C source files using the compiler's in-line assembly language capability. Formal specifications were written in Gypsy to take advantages of Gypsy's automated theorem proving. Development and test tools were written in a variety of languages, including C, Lisp, Icon, and Perl.

The source code size statistics were measured by examining the 'official' copies of all files comprising the LOCK system, stored in the LOCK configuration management system. The system, based on SCCS, stored LOCK source files in a special format that maintained a history of all modifications to it. The LOCK team customized SCCS to store a read-only 'official' copy of the latest version of every source file in the configuration management directory tree, nearby but separate from the corresponding SCCS repository file. The author used Unix text tools to measure the size of LOCK by analyzing the "official" copies of source files.

The code size measurements presented here are based on "Lines of Code" (LOC), that is, a count of the number of individual source code instructions contained in a unit of software. As described by Boehm [42], the number of source statements is closely correlated with the total effort required to develop a software component, and the effort per source statement is highly independent of language level.

Unfortunately, the LOC measurement is inherently approximate, since there is no universally accepted notion of what constitutes a single "source statement." Modern languages tend to allow relatively complex expressions in many places, and programmers can often embed groups of statements and even procedures inside a single, syntactic 'statement.' The measurements reported here were produced by counting the delimiters that often show up next to major expressions. To count C 'statements,' for example, the measurements counted the number of semicolons, closed brackets, and "#define" statements. This count reflects all source program statements and major computational expressions, and it omits blank lines and comments. The measurement requires nothing more than standard Unix text tools, which makes it relatively simple to compare these LOC counts with others' software.

Peer review played an important role in ensuring accurate measurements. Initial measurements in 1992 had simply measured lines of source code, including comments and blank lines. Internal reviews of the resulting measurements found them to be inconsistent with perceived sizes and productivity, so the measure-

ments were repeated, counting statements instead of lines. This second set of measurements produced the statistics presented here.

Unfortunately, the formal assurance team was not adequately involved in the review of source code sizes, and this led to measurement errors of the Gypsy specification files. To correct this, the author measured a later version of the same Gypsy specification produced by the same assurance team for the SMG project. This later specification was 5% larger than the previous version, both in line and character counts. The author counted the major expressions in those files (approximately one such expression for every three text lines), compared it against line and character counts, and cross checked the ratios to develop confidence in the result. The largest detectable discrepancy in the LOC estimates reported here for LOCK's Gypsy files was 5%. The resulting LOC estimates were peer reviewed by assurance team members.

Phase	Activity	Labor Hours	LOC	Language
All	Program Wide Activities	25,474	na	
	Hardware (build and buy)	8,176	na	
System Engineering	System Management	13,982	na	
	System Specifications	14,866	na	
Software: Design, Code, and Unit Test	Formal TCB Specifications	21,635	13,219	Gypsy
	Formal Lemmas		20,676	Gypsy
	TCB Kernel Software	36,703	42,202	C
	TCB Kernel Extension Software	10,801	28,607	C
	TCB SIDEARM Software	16,944	45,078	C
	LOCK/ix Kernel Software	18,250	62,663	C
	Ported LOCK/ix Utility Software		101,520	C
	Media Encryptor Software	19,490	41,638	C
Integration and System Testing	Integration, Test, Debugging	27,395	5,686	various
	Configuration Management	8,797	na	
Verification	Matching specifications to code	9,217	na	
	Formal TCB Verification	6,729	na	
	Cryptographic System Verification	6,582	na	
	Totals	245,041	361,289	

Table 4: LOCK program labor hours and lines of code

4.3. Productivity Categories

Table 4 organizes LOCK activities into five phases, and further sorts the LOCK WBS and source code statistics into seventeen categories. The phases correspond to rows appearing in Table 2. Individual categories indicate the amount of software created by the associated labor hours (measured in LOC) where appropriate.

Organizing the LOC information into categories was relatively straightforward, since the code was organized into well recognized components, and those components were stored in distinct directories.

Assignment of WBS tasks to productivity categories proved more difficult, if only because there were so many tasks to account for. To reduce the risk of inaccuracies here, the author followed a process that verified all category assignments. First, the author reviewed the WBS definitions associated with the activities to ensure that individual tasks were collected in categories that were likely to be correct. Next, the productivity category assignments and the resulting labor hour statistics were reviewed by senior members of the project, both technical and management, to ensure their correctness. Periodically the author has reviewed the category assignments and found better ways of structuring the data: this always involved consultations with individuals involved with the related tasks to ensure that the revised aggregation was correct. Following the initial data collection in 1992, the author went back and reviewed the data again in 1995, 1997, and 1999 in order to produce an accurate and comprehensible aggregation of data for publication.

The remainder of this section contains brief descriptions of the different productivity categories appearing in Table 4.

- Program Wide Activities

  This includes program management, program reviews, and program-wide planning and control. Also included are costs for user documentation produced under internal R&D.

- Hardware (build and buy)

  This covers the effort of selecting hardware to purchase and of constructing hardware from purchased components.

- System Management

  This includes the costs for system engineering management and high level planning for the LOCK program. Also included are the costs for certain system-wide documents, including the Systems Engineering Management Plan and the Software Development Plan, and trusted system manuals required by the TCESC.

- System Specifications

  This covers the development of the LOCK A- and B-level specification documents, along with related architectural issue analysis and resolution.

- Formal TCB Specifications

  This includes tasks directly associated with the development of the FTLS, DTLS, and FSPM. The LOC counts in this category refer to the size of the FTLS and FSPM coded in Gypsy. This is part of the TCB software “high assurance” cost shown in Tables 2 and 3.

- Formal Lemmas

  This category captures the LOC count of Gypsy ‘lemmas.’ The Gypsy formal specifications relied on a large set of lemmas that were necessary for the Gypsy theorem prover to automatically perform proofs based on the specification. The lemmas were not really necessary to express the formal specifications to human readers and reviewers. This is part of the TCB software “high assurance” cost shown in Tables 2 and 3.

- TCB Kernel Software

  This covers the detailed design, coding, and unit testing for the kernel software running on the LOCK host processor that is responsible for enforcing the LOCK security policy.

- TCB Kernel Extension Software

  This covers the detailed design, coding, and unit testing for additional trusted software running on the LOCK host.

- TCB SIDEARM Software

  This covers the detailed design, coding, and unit testing for the security decisionmaking software residing on the SIDEARM processor.

- LOCK/ix Kernel Software

  This covers the detailed design, coding, and unit testing for the LOCK/ix software that emulates standard Unix system calls while running in a restricted environment on LOCK. LOCK/ix development was reported as a single activity and was not broken down to show effort spent on kernel activities versus ported utilities. Discussions with the lead developer yielded an estimate that 85% of the effort was spent on kernel software and the remaining effort was spent on porting non-kernel utility software.

- Ported LOCK/ix Utility Software

  This covers the detailed design, coding, and unit testing required to make standard Unix utilities work under LOCK/ix. In some cases the utilities were modified to provide better access to LOCK's capabilities.

- Media Encryptor Software

  This covers the design, coding, and unit testing for the media encryptor software.

- Integration, Test, Debugging

  This covers system level integration, testing, and debugging. The LOC covered by this category are for development, testing, and support software not included in the delivered LOCK software components. Some fraction of the labor hours listed under this activity were used to implement that software. Much of this software was written in C, but portions were also written in Lisp, Icon, and Perl. We do not compute productivity rates for this software because the labor costs are not limited to software development activities.

- Configuration Management

  This covers planning, management, implementation, and support of the configuration management system used for LOCK.

- Matching specifications to code

  This includes the planning and initial implementation of the process for verifying the mapping between the LOCK specifications and its source code. This is part of the TCB software "high assurance" cost shown in Tables 2 and 3.

- Formal TCB Verification

  This covers the cost of proving consistency between the FSPM and the FTLS, of supporting the tools for machine proof checking, and of producing the covert channel analysis. This is part of the TCB software "high assurance" cost shown in Tables 2 and 3.

- Cryptographic System Verification

  This includes the tasks that produce media encryptor documents and analyses required to achieve official U. S. government cryptographic endorsement. This represents the media encryptor software's "high assurance" cost shown in Table 3.

5. LOCK Coding Productivity

LOCK code development consisted of four distinct efforts: TCB code, LOCK/ix code, media encryptor code, and formal specifications. While the latter effort didn't produce software that ultimately ran on the LOCK platform, its development was a key element in the LOCK program. If we look at pure coding productivity in terms of lines of code per hour, then all of the 'high assurance' coding activities yielded roughly sim-

ilar productivity metrics. The LOCK/ix effort, whose LOC count contains a lot of ported code, achieved significantly higher productivity when examined simply in terms of LOC on LOCK per unit time. Table 5 summarizes the LOCK coding effort in terms of labor hours, LOC, and average productivity.

LOCK TCB development involved typical operating system implementation activities. These included interrupt handling, device management, storage management, process management, and user management in addition to other features required of a TCB supporting multi-level security. Wherever possible, software was written to operate in user mode, including memory and mass storage management as well as device drivers. Productivity of 1.80 LOC per hour is reasonable for this type of work, particularly when the software had to be written to be consistent with a rigorous behavioral specification.

The media encryptor effort was similar to the TCB effort but achieved slightly higher software productivity at 2.14 LOC per hour. Both activities involved new software development with negligible amounts of ported software. Both activities developed software that needed to perform quickly and efficiently in a stand alone environment. Both needed to implement low level device control functions. However, the media encryptor implemented a relatively small number of functions compared to the TCB. Part of the productivity difference might also be attributed to differences between the media encryptor team and the TCB team: the latter was a larger team representing a broader range of development skills and experience, while the media encryptor team was smaller and more experienced.

LOCK/ix productivity was relatively high at 9 LOC per hour. Unlike the TCB and media encryptor, LOCK/ix was not a high assurance development activity. Also, the effort involved a significant amount of ported code. Of the 164,183 LOC comprising LOCK/ix source code, 101,520 LOC (62%) consisted of Unix utilities that for the most part were simply ported to the LOCK/ix kernel environment. Although the remainder was based on a portable Unix kernel, a significant amount had to be rewritten to make the most efficient use of LOCK functions without duplicating them in the LOCK/ix kernel. The lead LOCK/ix developer estimated that 85% of the 18,250 hours spent on LOCK/ix development was spent on the kernel software. This yields an estimated productivity of 4.04 LOC per hour for C code that required lower assurance.

Project Activity		LOC	% of LOC	Hours	Productivity (LOC/Hour)
C Source Code	TCB	115,887	32%	64,448	1.80
	Encryptor	41,638	12%	19,490	2.14
	LOCK/ix	164,183	45%	18,250	9.00
	Other	5,686	2%	na	na
Gypsy	Formal Specifications, Lemmas	33,895	9%	21,635	1.57
	Total	361,289		123,823	

Table 5: Coding productivity for different activities

Table 5 shows that the mathematicians developing the Gypsy specifications achieved a productivity of 1.57 LOC per hour. However, it is hard to interpret this information, since there is almost nothing reported in the literature on the LOC productivity of formal specification activities. It is interesting to note that it falls in the same general productivity range as other high assurance coding tasks.

The LOCK Gypsy code itself falls into two categories: the formal specifications and the lemmas. The formal specifications, comprising 39% (13,219 LOC) of LOCK's Gypsy code, represent the actual specifications associated with the LOCK platform to the level of detail deemed necessary to meet TCSEC A1 requirements. The lemmas were required so that the GVE could automatically check the consistency of the formal specifications. In order to make the proof checking work, mathematicians needed to produce numerous lem-

mas to properly link the policy elements with the design elements. These lemmas comprised 61% of the LOCK Gypsy code. In a practical sense, these lemmas were not really a part of the FTLS or FSPM. They were added to the specification to simplify the Gypsy proof checking process.

If we compare the size of the formal specification with the size of the TCB source code, we find that the specification is 11% the size of the TCB. The specification defines the behavior of the TCB interface, so this statistic could have different interpretations. For example, this statistic might reflect the level of detail of the specification, or perhaps it can predict the size of the TCB based on the size of its specification. This estimate omits the Gypsy lemmas, since they aren't strictly a part of the formal specification.

Although LOCK mathematicians seemed reasonably productive in terms of LOC per hour, the effort required to support automated proofs was judged to be excessive. Part of this complexity is reflected in the need to produce lemmas (61% of the Gypsy code) whose sole purpose was to automate the proof. In addition, the proof checking required a lot of computation that led to very lengthy execution times. Because of the excessive computation times, one mathematician used three separate Symbolics LISP Machines simultaneously to check three separate proofs. The Gypsy specification's total size also cast doubt on the correctness of the automated proof. There were some questions as to the soundness of Gypsy's proof checking, but the process was so long and laborious that it was impractical for mathematicians to double check the automated proofs manually.

The program achieved better success with "journal level proofs" designed to be checked by other mathematicians instead of machines[19]. These proofs were based on the FTLS with the Gypsy specific lemmas left out so that mathematicians could skip "obvious" steps in the proofs. This process was included in the LOCK assurance plan to address deficiencies with machine checked proofs as they had been used earlier. In particular, the journal level proof process seemed more likely to provide insight into why the system was or was not secure instead of simply providing a "yes/no" answer. This decision was consistent with well known arguments regarding the social role of proofs in establishing confidence in a particular result [43]. The principal shortcoming with journal level proofs was the concern that mathematicians might miss an error in the proof that the computerized checking would find.

6. LOCK Deployment: the Standard Mail Guard

By the time the LOCK follow-on work ended in 1991 the world of computing had radically changed. Multi-user timesharing hosts were replaced by desktop PCs linked to departmental network servers. ASCII terminals were obsolete, along with sophisticated security mechanisms mandated by the TCSEC to support them. LOCK was a solution looking for a problem.

Military organizations had embraced the new computing paradigm and were buying PCs, networks, and network servers at a furious rate. Almost all of these systems were standard, off the shelf commercial systems with no multi-level security capabilities. Organizations that processed classified information generally avoided operating in a multi-level mode. Instead, they established individual networks according to the security clearances of people in the department or organization, and kept networks separate according to their security levels. This introduced a number of practical problems since it made data sharing cumbersome and risky. Most organizations used a technique dubbed "sneakernet" which used removable disks and tapes to transfer data between systems operating at different security levels. Some organizations introduced guard systems that sat at the boundary between networks at different security levels and allowed them to exchange data in highly restricted circumstances. Typical guards were designed to pass highly formatted data whose security properties could be automatically analyzed and verified.

By mid-1992, LOCK supported TCP/IP networking and had demonstrated an ability to handle Internet electronic mail. The TCP/IP protocol stacks resided on separate network interface boards, one for each network security level. This allowed the protocol software to run in parallel with other LOCK processes and isolated the protocol software from the LOCK TCB. As a demonstration, LOCK could receive an e-mail

message from a network at one security level and deliver it on a network at a different level. This capability made LOCK a very promising platform for building a guard that handled electronic mail.

The Standard Mail Guard (SMG) took this capability and enhanced it to produce a practical device for exchanging e-mail between networks at different security levels, particularly between authorized users on sensitive but unclassified government networks and authorized users on networks classified Secret [22]. To prevent the leakage of secret information to the unclassified network, users on the secret network could only send a message after they had manually reviewed it to verify its unclassified contents.

The transition from the LOCK concept demonstration to an operational SMG took approximately eighteen months. Although exact statistics are not available, approximately half of the development effort was spent on LOCK reliability and performance improvements. One quarter was spent on new software development to handle message filtering and user administration features. The remaining effort was spent on system testing.

When it arrived for initial deployment, the SMG was dramatically different from the original LOCK platform. The CPU was upgraded to the MC68030, the separate housings for the LOCK host and the SIDEARM were merged into a single chassis. Most important, the system operated for weeks under load without crashing – an essential feature of any device in operational use.

Before its initial deployment in mid 1994, formal assurance for the SMG consisted of a general update of the existing LOCK formal specifications to reflect minor interface changes. Since the networking software resided almost entirely outside the TCB, it had little effect on the FTLS. Policy statements in the FSPM were updated to reflect some basic implications of the SMG applications, and the FTLS was updated to incorporate the guard's kernel extensions. It is important to note that the SMG application software simply bypassed LOCK's multi-level security. This was inevitable because of the nature of guard applications. Unfortunately, it rendered the formal analysis irrelevant. The formal analysis asserted that SMG implemented multi-level security correctly except when it was bypassed by privileged software. The policy exempted application software from formal assurance requirements and assumed that the application would be validated by some other, unspecified process. In later releases the SMG policy was extended to specify how the guard was supposed to work, which eventually made the proof results relevant to the system's security.

The bulk of the assurance effort for the SMG was system testing. Hundreds of tests were developed and run in the months preceding its releases. This included a large set of tests based on the top level descriptions to ensure that the code was consistent with the assumptions built into the formal specifications. There was also an exhaustive test of domain and type access permissions defined in the type enforcement mechanism. This test invoked subjects in all domains to access all other domains and types in order to show that access permissions defined in the type enforcement specifications were actually enforced by the implementation. There was also a similar test of multi-level security. To test network security, SMG was subjected to a set of Internet penetration tests that exercised all known weaknesses of Internet protocol software. The team also developed an extensive acceptance test incorporating 89 separate tests that required over 500 hours of run time.

In late 1994, Secure Computing announced Sidewinder, a commercial Internet firewall. Traditional firewalls tended to fall into one of three architectures, in order of increasing security: packet filters, circuit relays, and application gateways [40]. Sidewinder implemented an application gateway architecture using type enforcement technology and the assured pipeline concept [24]. This was essentially the same architecture as the SMG, repackaged in a commercial product.

7. Flaw Detection

An essential question to ask is whether formal assurance is cost effective: given finite project resources, what is the best way to expend effort on flaw detection? While the LOCK and SMG programs give no conclusive answers, they illustrate both the power and limitations of formal methods.

According to the TCSEC, high assurance techniques should yield "a high degree of assurance that the TCB is correctly implemented." In an ideal world, a "correctly implemented" system is bug free; in the real world, we might reasonably expect such a system to contain as few flaws as possible. Here, we will look at how well formal assurance served the job of detecting flaws from two different directions. First, we look at the LOCK experience, which illustrates how well formal assurance can locate flaws in the components it models. Second, we look at the SMG experience, in which we see the narrowness of the intense scrutiny achieved by formal assurance.

7.1. The LOCK Experience

LOCK assurance was driven by the TCSEC, and the TCSEC treated multi-level security as the essential capability that must be implemented without flaws, leaks, or covert channels. Both formal and less formal assurance requirements focused on the correct behavior of multi-level security mechanisms. As shown in Figure 2, formal methods only addressed part of the overall assurance picture: at best they proved that the design could successfully enforce the modeled features, specifically the multi-level security constraints.

Table 6 shows what percentage of the multi-level security flaws found in LOCK were uncovered by the different assurance processes. Formal methods accounted for the vast majority of security flaws detected: 68 percent. The "Effort" column examines the relative amount of effort spent on different assurance processes. A relative comparison of flaw detection rates to effort can illustrate the relative efficiency of different assurance activities at finding flaws. The most interesting result is that the formal methods do well overall in detection efficiency as well as detection rate.

However, these observations are subject to caveats, because of assurance process interdependence and completeness. Ideally, we might like to pick and choose the most efficient detection method and skip the others. If the formal proofs by themselves found 25% of the flaws while using only 8% of the assurance effort, then wouldn't it be efficient to spend most of our time on that? Of course, it's impossible. We can't do the formal proofs unless we've spent the money on the formal specifications. Moreover, some of the table's results are undoubtedly misstated due to incompleteness. As noted earlier, the LOCK program only matched specifications to code for a portion of the TCB; the detection rate would most likely increase as more of the TCB's code is covered by it. This suspicion is borne out by studies of code inspection in general, which have found it to be very effective at flaw detection [42]. Arguably, the LOCK testing effort was likewise incomplete. LOCK's testing activity was significantly less extensive than on the subsequent SMG program. The LOCK program's initial objective was research, and formal assurance carried a higher priority than testing when competing for finite program resources. For this reason, the LOCK program never developed or performed a "level test" that systematically tested all accesses possible for a predefined set of security levels. Such tests on SMG uncovered numerous flaws in the multi-level security mechanisms when the tests were first developed and run.

Phase	Assurance Process	MLS Flaw Detection by Process %	Effort	
			Labor Hours	%
System Engineering	System Specifications	21%	14,866	19%
Software	Formal TCB Design Specifications	25%	21,635	27%
Verification	Matching specifications to code	18%	9,217	12%
	Formal TCB Verification (proofs)	25%	6,729	8%
Integration	Integration, Test, Debugging	11%	27,395	34%

Table 6: How multi-level security (MLS) flaws were detected on the LOCK program

Figure 1 showed how the assurance steps ran in parallel with its software development steps; this happened on for the LOCK and SMG programs. This was caused by real world constraints on cost, schedule, and available personnel. It wasn't feasible to assemble a large staff of mathematicians to construct the formal assurance models completely and still have the calendar time to finish the software development effort within the required time period. As the LOCK program proceeded, management was aware of the way that lengthy delays could lead to project cancellations. Moreover, given how the duration of the LOCK program had spanned the end of timesharing and the flourishing of networked workstations, the SMG team took such concerns very seriously. This led to the decision to run the processes in parallel.

This decision ensured that the programs shared a common experience of other large scale software development projects: late changes that affected requirements, design, and source code. While an idealized development process based on formal design assurance might be able to avoid the problems wrought by such changes, such problems might simply be inevitable in large, real world projects. Changes in the LOCK program rarely came from external sources, since LOCK requirements were based on the published and unchanging contents of the TCSEC. Instead, changes emerged from flaws detected by formal assurance activities. Flaws detected while analyzing LOCK requirements and developing its formal security policy were very timely, since they were detected before software design and coding occurred.

However, the FTLS was developed at the same time as the source code, so flaws detected at that point occasionally caused changes in the software design and occasionally affected existing source code. Formal proofs were performed at the same time as system testing, after coding was essentially complete. Flaws found by the formal proofs tended to be more troublesome than flaws found by testing, since the proofs detected requirements and design flaws, necessitating larger changes to the system baseline. However, such problems did not cause major dislocations within the project; they were simply a reminder that LOCK was a large software project, regardless of whatever else it might have been.

7.2. SMG Experience

As noted earlier, the SMG augmented its formal assurance methods with an extensive testing program that far exceeded what was performed under the LOCK program. Tests covered the gamut of strategies and motivations: requirements based, design based, vulnerability, penetration, performance, and stability testing. As the program progressed to later releases, the tests increased in sophistication, particularly the internal systems tests that preceded the government's acceptance testing. In particular, major improvements were made in the sophistication of penetration tests and in specification based testing that verified the top level descriptions of TCB functions.

SMG Flaw Statistics (~ = estimate)	90 day study	% of flaws	First release total	First release extrapolation
Total Flaws Reported	~205		1,531	1,531
Flaws inherited from LOCK (deduct from total)	na		~300	~300
Flaws detected by the assurance team	39	19%		~234
Flaws in multi-level security	24	12%		~144

Table 7: SMG Problem Reports: deployment status and 90 day study

Table 7 summarizes some flaw statistics from the SMG program. Numbers marked with a tilde (~) indicate approximations. There were 1,531 flaws detected while developing the first release of the SMG in 1994. Of those, approximately 300 flaws, all functional flaws as opposed to multi-level security flaws, were flaws detected earlier in LOCK and "inherited" by the SMG program. The "90 day study" refers to a study of SMG flaws detected by members of the formal assurance team during a 90 day period. During that same

period, it was estimated that approximately 205 flaws were detected in the SMG by the development team, the test team, and the assurance team together. As with the LOCK program, formal assurance found the lion's share of flaws in multi-level security.

While overall flaw detection statistics weren't available for LOCK, those statistics for SMG tell an interesting story: formal assurance efforts accounted for only 19% of all flaws found. At the time of the 90 day study, assurance accounted for one-half to one-third as much project effort as testing, but only found a fifth of the errors. The low detection rate is of course because only 12% of all flaws detected involved multi-level security. The assurance team would occasionally find other types of errors, but they weren't explicitly tasked or trained to do so.

While it's true that a security conscious end user won't want to deploy a "secure" platform if it might have flaws in its multi-level security mechanism, the user likewise won't deploy a platform that's unreliable. There is no way to know if testing alone would have detected all significant security flaws in the SMG, that is, all flaws that might in fact compromise classified information. We also don't know how many flaws found by formal methods would have been missed by testing, whether significant or not. However, the formally assured LOCK platform proved too unreliable for serious use, while the throroughly tested SMG was deployed world wide in a critical application. Testing was essential, since it's not appropriate to expect a security-oriented formal assurance process to improve a system's availability, stability, or performance.

Moreover, the SMG flaw detection statistics suggest that formal assurance team was much less "productive" at finding flaws than the testing team. The smaller assurance team found a lot fewer flaws than their size and level of effort might otherwise lead us to expect. In a sense, this result is especially important when one considers that assurance teams and test teams are very different. Test teams are comparatively easy to build and staff. Any high technology community will have a large population of experienced testers, and beginning testers can be productive with limited training. Formal assurance teams, on the other hand, are rare and very difficult to build. Team members require an unusual aptitude for mathematics, and productive team members generally have a graduate degree. Team leaders with experience in formal assurance projects are very rare in comparison to test team leads. This represents a set of costs that aren't captured in these statistics, but certainly affect the cost-effectiveness of formal assurance.

The 90 day study provided an additional piece of relevant information: it indicated which processes the assurance team used to detect security flaws. As in the LOCK program, the formal specification and proof processes were very effective in finding flaws. The SMG specification process was responsible for finding 28% of the MLS flaws detected (compared to 25% on LOCK). The verification and proof process was responsible for finding 23% of the flaws detected (compared to 25% on LOCK). Thus, measurements from both projects credit these formal techniques with finding at least half of the critical security flaws detected.

8. Summary

The LOCK program tried to explore the issues of high assurance, and the SMG applied the resulting system to a high security problem. The project results yield the following observations:

- High assurance required a large commitment of labor hours, both for formal assurance methods and for the rigorous, though less formal, methods required for cryptographic endorsement. On the LOCK program, the tasks associated specifically with high assurance added a 26% premium to the labor costs for software development.
- The extra labor required for high assurance can also translate into calendar time. To reduce the effects of high assurance on the project schedule, both LOCK and SMG performed software development in parallel with some of the later steps in the formal assurance process. This did not cause any 'show stopping' problems; at most it treated the programs to the type of requirements changes that often infect other large scale software development projects.
- When looking purely at TCB labor costs, the extra tasks required to comply with TCSEC A1 formal assurance requirements added 58% to the labor costs. While this may overstate the relative

cost somewhat (since it doesn't factor in certain system-wide costs) it understates the total A1 cost, since LOCK did not complete the A1 evaluation process despite all the time and money spent.

- If we compare the ability of formal assurance to detect flaws with that of a vigorous testing program (the SMG experience), we find that formal assurance is less 'productive' in terms of the number of flaws found within a given number of labor hours. When we combine this observation with the real world problem of trying to build a formal assurance team, the costs of formal assurance will outstrip the resources of most software development projects.
- Formal assurance was very effective at finding the flaws it was designed to look for: errors in the design of the multi-level security mechanism and related mechanisms. Statistics from both the LOCK and SMG program show that the vast majority of multi-level security flaws were found using formal methods.

This history of LOCK and SMG gives a somewhat positive but mixed review of the costs and benefits of formal assurance. As we look forward, there are observations and lessons that might profitably be applied to future projects using formal assurance:

- Some of the shortcomings that limited the sophistication of LOCK and SMG specifications could probably be overcome with modern specification tools. Today's formal assurance tools have dramatically improved over the GVE of the early 1990s. LOCK assurance personnel have used modern tools on subsequent projects, and have found them to be vastly superior [44]. Modern specification languages are more refined and the automatic provers have far more effective automation, making the entire process more effective. Moreover, the modern, faster processors make the proof process much more efficient.
- Formal assurance techniques can make an appropriate and cost-effective contribution to flaw detection, but only if the formally specified requirements cover the system's critical requirements thoroughly enough. Formal assurance techniques are very good at finding flaws, and they achieve an aura of completeness that testing can never match. Unfortunately, the techniques aren't useful if the formally analyzed requirements do not cover enough of the system's critical requirements.
- A challenge posed by the LOCK and SMG work is to establish criteria to determine whether or not formal assurance would be cost-effective for a particular project. A project's budget must be allocated to the most effective assurance techniques, and that depends on features of the project itself. There need to be better criteria or even heuristics for identifying projects would profit from formal assurance.
- Several of the author's colleagues have moved from formal assurance of operating system security to formal assurance of integrated circuit operation, which has apparently emerged as a practical application of these techniques. No doubt there are other applications where formal assurance can make a practical contribution.

8.1. Acknowledgments

This paper describes work performed for the National Security Agency under contract MDA904-87-C-6011 and contract MDA904-92-C-2094.

Some of this material was distilled from presentations and internal reports on LOCK and SMG by W. Earl Boebert, Dr. J. Thomas Haigh, Dr. Spence Minear, Todd Fine, T. M. Markham, Dr. Richard C. O'Brien, and Dr. Carol Muehrcke. Scott Hammond provided essential insight into LOCK/ix development. Dr. John Hoffman provided information and interpretation regarding the SMG 90 day flaw study. Todd Fine performed a crucial review and critique of an earlier version of this paper. Bill Erbes helped clarify the background of the DFARS, and John Applegate helped unearth some prehistoric DOD specifications to help finish the story. The author also sincerely thanks the anonymous reviewers for their detailed comments and recommendations.

This information was collected, analyzed, and prepared for publication because my managers at the Advanced Technology Division at Secure Computing Corporation considered it a worthwhile exercise: Chris Filo, Vice President for Advanced Technology, Don Nutzmann, Dr. John Hoffman, and Dr. Richard C. O'Brien.

8.2. References

1. Willis Ware, "Security Controls for Computer Systems (U): Report of Defense Science Board Task Force on Computer Security," Rand Report R609-1, The RAND Corporation, Santa Monica, CA (February 1970).
2. James P. Anderson, "Computer Security Technology Planning Study," ESD-TR-73-51, ESD/AFSC, Hanscom AFB, Bedford, MA 01731 (October 1972) [NTIS AD-758 206]
3. DOD, "Security Requirements for Automatic Data Processing (ADP) Systems," DOD Directive 5200.28, revised April 1978.
4. C. Weissman, "Security Controls in the ADEPT-50 Time-Sharing System," *Proceedings of the 1969 Fall Joint Computer Conference*, AFIPS, 1969.
5. Dorothy E. Denning, "A Lattice Model of Secure Information Flow," *Communications of the ACM*, vol 19, no 5, May 1976, pp 236-243.
6. Ravi Sandhu, "Lattice-Based Access Control Models," *IEEE Computer* vol 26, no 11, November 1993, pp 9-19.
7. David E. Bell and Leonard La Padula, "Secure Computer System: Unified Exposition and Multics Interpretation," ESD-TR-75-306, ESD/AFSC, Hanscom AFB, Bedford, MA 01731 (1975) [DTIC AD-A023588]
8. Butler Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, vol 16, no 10, October 1973.
9. DOD, "Trusted Computer System Evaluation Criteria," DOD-5200.21-STD, December 1985.
10. Santosh Chokhani, "Trusted Products Evaluation," *Communications of the ACM*, vol 35, no 7, July 1992, pp 64-76.
11. Richard E. Smith, "Trends in Government Endorsed Security Product Evaluations," *Proceedings of the 23rd National Information Systems Security Conference*, Baltimore, MD, October, 2000.
12. DOD Computer Security Center, "Computer Security Requirements -- Guidance for Applying the Department of Defense Trusted Computer System Evaluation Criteria in Specific Environments," CSC-STD-003-85, 25 June 1985.
13. L. Fraim, "The Challenge After A1: A View of the Security Market," *Proceedings of the 9th National Computer Security Conference*, National Computer Security Center, September 1986, pp. 41-46.
14. S. Lipner, "Secure System Development at Digital Equipment: Targeting the Needs of a Commercial and Government Customer Base," *Proceedings of the 8th National Computer Security Conference*, National Computer Security Center, September 1985, pp. 47-54.
15. Timothy E. Levin, Steven J. Padilla, Roger R. Schell, "Engineering Results from the A1 Formal Verification Process," *Proceedings of the 12th National Computer Security Conference*, National Computer Security Center, October 1989, pp. 65-74.
16. Gary R. Stoneburner and Dean A. Snow, "The Boeing MLS LAN: Headed Towards an INFOSEC Security Solution," *Proceedings of the 12th National Computer Security Conference*, National Computer Security Center, October 1989, pp. 254-266.
17. Tim Leonard, Compaq Corporation (nee' Digital Equipment Corporation), personal communication.
18. O. Sami Saydjari, Joseph K. Beckman, and Jeffrey R. Leaman, "LOCKing Computers Securely," *Pro-*

ceedings of the 10th National Computer Security Conference, National Computer Security Center, 1987.

19. O. Sami Saydjari, Joseph K. Beckman, and Jeffrey R. Leaman, "LOCK Trek: Navigating Uncharted Space," *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, Oakland, CA, 1989.
20. Peter G. Neumann, L. Robinson, Karl N. Levitt, R. S. Boyer, and A. R. Saxena, "A Provably Secure Operating System," M79-225, Stanford Research Institute, Menlo Park, CA 94025 (June 1975)
21. W. E. Boebert, R. Y. Kain, and W. D. Young, "Secure Computing: The Secure Ada Target Approach," *Scientific Honeyweller*, June 1985.
22. Richard E. Smith, "Constructing a High Assurance Mail Guard," *Proceedings of the 17th National Computer Security Conference*, National Computer Security Center, 1994.
23. Dan Thomsen and Winn Schwartau, "Is Your Network Secure?" *Byte*, January 1996.
24. Richard E. Smith, "Sidewinder: Defense in Depth Using Type Enforcement," *International Journal of System Management*, 1996.
25. National Computer Security Center, "Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria," NCSC-TG-005, Version 1, 31 July 1997.
26. Mark A. Schaffer, and Geoff Walsh, "LOCK/ix: On Implementing Unix on the LOCK TCB," *Proceedings of the 11th National Computer Security Conference*, National Computer Security Center, 1988.
27. AIM Technology, "AIM Technology Published Price List," AIM Technology, Santa Clara, CA, 15 March 1988.
28. R. Y. Kain, "Throughput Benchmarks with AIM," memorandum to LOCK team in LOCK program archives, 1 August 1988.
29. W. E. Boebert, and R. Y. Kain, "A Practical Alternative to Hierarchical Integrity Policies," *Proceedings of the 8th National Computer Security Conference*, National Computer Security Center, 1985.
30. R. C. O'Brien, and C. Rogers, "Developing Applications on LOCK," *Proceedings of the 14th National Computer Security Conference*, October 1991.
31. D. D. Clark, and D. R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1987.
32. W. Earl Boebert, "Annotated TCSEC," Honeywell Secure Computing Technology Center, 1988.
33. DOD, "Specification Practices," MIL-STD 490A, 4 June 1985.
34. D. I. Good, A. L. Ambler, J. C. Browne, W. Burger, R. Cohen, C. Hoch, R. Wells, Gypsy: A Language for Specification and Implementation of Verifiable Programs, In *Proceedings of ACM Conference on Language Design for Reliable Software*, March, 1977.
35. D. I. Good, R. M. Cohen, C. G. Hoch, L. W. Hunter, and D. F. Hare, "Report on the language Gypsy, Version 2.0," Inst. Comput. Sci., Univ. Texas at Austin, Tech rep. ICSCA-CMP-10, Sept. 1978.
36. J. Thomas Haigh, and William D. Young, "Extending the Noninterference Version of MLS for SAT," *IEEE Transactions on Software Engineering*, vol SE-13, no 2, February 1987.
37. National Computer Security Center, "Introduction to Certification and Accreditation," NCSC-TG-029, Version 1, January 1994.
38. Tad Taylor, "FTLS-Based Security Testing for LOCK," *Proceedings of the 12th National Computer Security Conference*, October 1991.
39. M. J. Rochkind, "The Source Code Control System," IEEE Transactions on Software Engineering, vol SE-1, (December 1975), pp. 364-370.
40. "Department of Defense Federal Acquisition Regulation Supplement," 48 *Code of Federal Regulations*, Chapter 2, Part 234.

41. William R. Cheswick and Steven M. Bellovin, *Firewalls and Internet Security*, Reading, MA: Addison-Wesley, 1994.

42. Barry Boehm, *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall, 1981.

43. R. A. DeMillo, R. J. Lipton, and A. J. Perlis, "Social Processes and Proofs of Theorems and Programs," *Communications of the ACM*, vol 22, no 5, May 1979.

44. Todd Fine, personal communication.