RC24455 (W0712-054) December 14, 2007
Computer Science

IBM Research Report

Tooling in Support of Common Criteria Evaluation of a High Assurance Operating System

David C. Toll, Sam Weber, Paul A. Karger,
Elaine R. Palmer, Suzanne K. McIntosh
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Tooling in Support of Common Criteria Evaluation of a High Assurance Operating System

David C. Toll, Sam Weber
Paul A. Karger, Elaine R. Palmer, Suzanne K. McIntosh

IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598-0704, USA

Intended Audience

This document targets software developers and software development project managers who are considering undertaking or planning a high assurance Common Criteria evaluation. Some familiarity with Common Criteria is assumed.

Scope

This document discusses productivity tools that facilitated our efforts to create a high assurance operating system for smart card chips and evaluate it under the Common Criteria at EAL7. It does not discuss the features of the operating system itself. A detailed description of the operating system can be found in [39].This document includes only a brief discussion about Common Criteria at high assurance levels; however, further details can be found in [4-6].

Organization of this Article

This document has three major parts: the technical motivation behind our work, the lessons we learned, our suggestions for tools for anyone courageous enough to undertake such an effort in the future, and a summary of our findings. A final section discusses the impact of the project, its current state, and anticipated future work.

Technical Motivation

Project Background and Goals

In 1998, a team at the IBM Thomas J. Watson Research Center embarked on a project to develop a high assurance operating system for smart cards, which could support multiple, field-downloadable, native applications (written in C and Assembler) from arbitrary (potentially hostile) sources. The operating system took advantage of the first smart card processors with hardware protection features, and enabled verifiable protection of the operating system from applications, and applications from each other. From the outset, the system was designed to meet the highest evaluation assurance level defined under the Common Criteria, namely EAL7.

Unique features of the operating system include a formally specified, mandatory security policy that provides multilevel security. Target applications include those suitable for governments as well as commercial enterprises. In order to enforce a mandatory security policy, strong authentication is required. Thus, the operating system also contains a privacy-preserving, two-way authentication protocol integrated with the mandatory security policy. The authentication protocol relies on a cryptographic library, which was certified separately under the Common Criteria at EAL5+ for use with other systems.

Currently emerging applications now validate our envisioned need for a high assurance operating system for smart cards. For example, electronic visas and identity cards for multinational coalitions could benefit from such a platform, if it were commercially available. While our initial platform was smart cards, the operating system design is applicable to other systems with limited memory, such as cell phones, USB tokens, and PDAs. Requirements cited by the next generation of SIM cards for mobile phones explicitly describe features and security provided by our design. The next generation of mobile phones will be expected to run multiple simultaneous financial applications, as well as video applications requiring strong digital rights management and security isolation [9]. The applications of this technology are not limited to small devices. The mandatory security policy has been applied to systems running on large servers [13] [35].

The operating system was named “Caernarvon”[1] after the magnificent castle in North Wales, which was built at the end of the thirteenth century and is still standing. Construction of the original Caernarvon lasted 47 years [26]. Construction of its electronic namesake was also expected to be an arduous and lengthy development effort. We divided the project into six nearly equal efforts: design, implementation, test framework / testing, Common Criteria documentation, formal modeling, and vulnerability / covert channel analysis.

Initially, the project team spanned six enterprises: IBM, Philips Semiconductors (now NXP Semiconductors), atsec Information Security, the German Federal Office for Information Security (BSI), the German Research Center for Artificial Intelligence (DFKI), and a Common Criteria evaluation laboratory in Germany. We found it absolutely necessary that the enterprises have legal agreements in support of full disclosure of technical details because of the many security-sensitive interactions among the hardware, software, formal model, Security Targets, and evaluation technical reports. The technical rationale behind this finding is given in [22].

Common Criteria

The Common Criteria [4-6] is an ISO standard (ISO 15408:2005) for evaluation of the security aspects of a computer system. It evolved from earlier evaluation criteria, including the US Trusted Computer System Evaluation Criteria (TCSEC) [7] and the European Information Technology Security Evaluation Criteria (ITSEC) [8]. The Common Criteria requires an independent third-party evaluation of the product, examining both the security functional requirements and the evidence, dependent upon the assurance level, that the functional requirements are actually correctly implemented. The evaluation assurance levels (EAL) are measured from EAL1 (lowest) to EAL7 (highest). Typical commercially available operating systems have received Common Criteria certificates at EAL4. Levels EAL6 and EAL7 are often called the *high assurance* levels, because systems evaluated at those levels under the Common Criteria (or equivalent levels under the TCSEC or ITSEC) are the only systems that have been shown to be generally resistant against sophisticated penetration attacks. These attacks are currently commonplace but used to be considered only the concern of the military [12, pp. 7-19], and include such problems as buffer overflows, incomplete argument validation, spyware, Trojan horses, and root kits. High assurance is specifically designed to address these problems.

To pass an EAL7 evaluation, the Common Criteria requires the strongest software engineering techniques known. These techniques include a formal security policy model, a full system design with a formal high-level design specification, and a formal proof of correspondence between the security policy model and the high-level design specification. It also requires a specification of all internal functions in a semi-formal low-level design with a demonstration of correspondence between the high-level design, the low-level design, and the actual software code. The development cycle must use well-defined tools and must include intensive design, code reviews, and full configuration control. There must be comprehensive testing, including code coverage analysis that every path has been exercised and that no dead code exists. Finally, there must be an extensive vulnerability analysis for possible security loopholes, as well as a search for covert communications channels.

Smart Card Platform Limitations

Smart card implementations are constrained by the very same limitations imposed on designers and developers of firmware for other types of embedded systems. In the subsections that follow, we describe some of the major limitations experienced when developing for smart card platforms.

Memory Limitations

In smart card platforms, the amount of available RAM, EEPROM, and ROM is significantly limited. In current smart card processors, typical RAM size is 2K to 6K bytes, while typical EEPROM size is 32K to

[1] In Welsh it is pronounced approximately “kire-NAR-fon”; in English, “KAR nar vən.”

144K bytes, and ROM size is 64K to 256K bytes. The constraints are not surprising, since smart card chips normally are used in packaging no thicker than a credit card. The memory available in a smart card chip is markedly smaller when compared to non-embedded programming environments, where virtual memory capabilities provide the developer with ample memory space.

The stringent memory limitations found in embedded environments create a situation where developers must sparingly use both data and program storage by carefully crafting hand-tailored algorithms—often writing in assembler language rather than in a higher level language. It is important to ensure that coding in such a manner results in code and data structures that are easy to understand, debug, and maintain. Unless a development organization has strict guidelines to ensure that the code is easily read by humans and that design decisions and tradeoffs have been documented, it can be difficult for new members of the team to become productive.

Limitations of Smart Card Development Tools

The main tools for developing software in both embedded and non-embedded environments include compilers, linkers, integrated development environments (IDEs), emulation systems, and simulators.

In the embedded space, these tools often have limited functionality when compared to tools available for non-embedded development environments. The tools may have only a small user population. It is common for users to find problems that have not been encountered by others.

The limitations of the tools lead users to develop home-grown extensions and compensatory solutions. Once developed, these additional tools are not made available to the user community, but instead companies find a competitive advantage in keeping them proprietary. Hence, each consumer of the tool is faced with maintaining the custom extensions.

Performance Limitations

Somewhat surprisingly, we ran into few performance limitations with smart card chips. While the speed of the chips is much lower than modern chips used in PCs or workstations, the speed is generally quite adequate for typical smart card applications. Most modern smart card chips include hardware crypto accelerators, so even public key cryptographic operations can be carried out with sufficient speed. The biggest performance limitation that we encountered was not with the smart cards themselves, but with contact smart card readers. Many of the contact readers that have been deployed over the years limit the communications speed to only 9600 baud, even though modern smart card chips can both process and communicate at much higher rates. The 9600 baud limitation on contact smart card readers can cause apparent performance problems with public-key authentication protocols because of the time required to transmit certificates to and from the card. These transmission times are much longer than the time needed to actually apply or check digital signatures. With either faster contact readers or contactless readers (which provide much faster data transfer rates), the performance problems disappear.

Lessons Learned

Design and Implementation Documentation

A product that is designed to be evaluated at a high level under the Common Criteria must necessarily have complete documentation. The Caernarvon project from its inception had a full specification and documentation, which was written using FrameMaker.[2] FrameMaker is a manual production program that is widely used in industry for technical documentation—it provides facilities to generate a set of documentation that comprises multiple books, with tables of contents, indexes, cross references both within and across volumes, etc. The Caernarvon documentation was written using templates for complete books, including the cover, title page, table of contents, preface, chapters, appendices, bibliography, index, etc., that produce documentation in a format used for IBM manuals. We relied heavily on the robustness of FrameMaker, using thousands of cross references that spanned multiple large volumes totaling several thousand pages. The documentation was produced in the form of PDF files, with all cross

[2] FrameMaker is a trademark of Adobe Systems Incorporated.

references (including cross-volume references), index entries, tables of contents entries, etc., appearing as hypertext links.

The project coding standards required that the specification of each function be included in comments at the start of the function. The Common Criteria also requires that this specification be included in the low-level documentation of the system submitted for the evaluation. Hence it was decided that much of the low-level documentation for the code (and the test suite) would be generated from the source code, rather than being written twice (once in the code, and once in the documentation). We also desired that the low-level documentation be consistent with the rest of the system documentation (in the same style and in the same sequence of volumes, for example, and enabling cross-references both into and from the generated low-level documentation). FrameMaker files normally are held in a proprietary, binary format; however, FrameMaker also has a text form of file named Maker Interchange Format (MIF) files, the documentation for which is included in the FrameMaker distribution. MIF files can be generated by external tools, and these files can then be embedded into the books.

Generating FrameMaker documentation files from the source required a tool to process the source files and output MIF files so that the generated documentation had the same style and format as the other volumes. We first experimented with *Autodoc*, an IBM-internal tool for extracting documentation from C code. Next, we considered the Ruby "standard" facility (called *RDoc*) to produce documentation from Ruby source, which generates HTML documentation or documentation that can be read using a command line program named *ri*. However, we rejected this choice because the generated files are not MIF files and are incompatible with the other documentation of the system. Further, this program did not work on Assembler code. As a result, we developed a new program in Ruby, which processed C, Assembler, and Ruby source files and generated MIF files. The resultant documentation was easily readable in its source form in the program files, even with the necessary control tags included. The generated files were fully compatible with the remainder of the documentation, including cross-references and index entries, inserted tables and figures, and generated tables. We also created full documentation of this Ruby program, which was necessary to meet the Common Criteria requirement that only well defined tools be used.

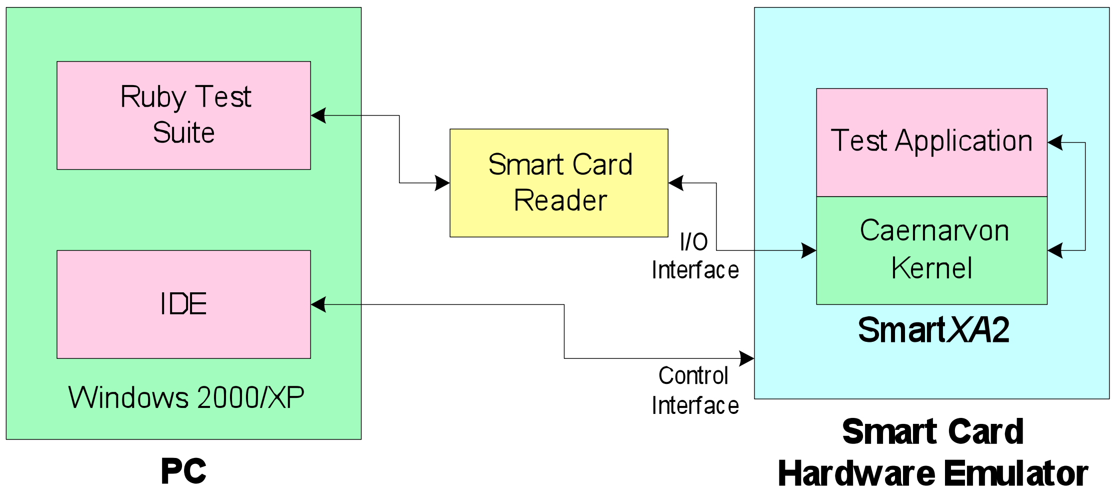
In conclusion, we have a comprehensive and professional set of documentation, with a consistent style, which includes the system specifications, the low-level code documentation, and the test suite documentation. The cost of this was that a fair amount of effort was expended in extending existing tools or developing new ones to provide the facilities we needed to accomplish this goal.

Development and Test Tools

As mentioned in the section "Limitations of Smart Card Development Tools" above, there is a limited market for development tools for a processor such as the Smart*XA*2, and as a result the supported tool suite for the development and testing of code for the Smart*XA*2 is available from only one company. The tool set includes the required C compiler, assembler, and linker, plus a few other utilities (the system was written mostly in C, with a small amount of assembler). These programs are run from an IDE that has a graphical interface and provides additional facilities such as program editing. This complete development environment runs under Windows 2000 or XP.[3] The tool set also comes with a hardware emulator, running a special "bond out" (i.e., debug) version of the chip, which has EEPROM memory instead of the ROM and can be controlled from the IDE. Code compiled and linked under the IDE is downloaded into the emulator and can be run under the control of the debugger within the IDE, using break points, to examine the contents of variables, memory and registers, etc.

The smart card system under test also has its "normal" I/O interface, which is a special form of serial interface. This communicates with a smart card reader, which in turn communicates with a computer via a "regular" serial interface. This interface is used to communicate with the code running within the smart card, which is the system under development running in the emulator. The arrangement of the testing environment, with its two interfaces, is shown in the following diagram.

[3] Windows 2000 and Windows XP are trademarks of Microsoft Corporation.



This diagram shows the IDE, which includes the program development tools, running under Windows XP, and the control interface from the IDE to the emulator. Also shown is the separate test suite (see below), which executes test scripts and sends appropriate commands to the smart card system under test via the smart card reader and the smart card's I/O interface.

There are many varieties of smart card readers available for purchase, but many of them have the property, undesirable here, that they implement much of the low-level communications protocol with the smart card. Further, there is "standard" software available within Windows that will communicate with smart cards; unfortunately, again this performs much of the low-level communication. In a development environment such as this, simple transparent reader and driver software is required to enable control, from the test environment, of all of the communication with the smart card. For this reason, we used IBM smart card readers, which could be re-programmed, together with a locally written DLL, to enable the Ruby test suite to communicate with the reader.

The Common Criteria requires the use of a configuration management system (CMS) to store the implementation, the system documentation, all tools, all tests, lists of common commercial software used, and various other project-related documents. Furthermore, it is common practice in a development project such as this one, involving numerous parties in several locations, for the files of the project to be stored in a CMS. We considered using Microsoft's Visual SourceSafe (VSS) system. However, the software license agreement at that time included the restriction that it be used for "purposes of designing, developing, and testing your software product(s) that are designed to operate in conjunction with any Microsoft operating system product." Unfortunately, the Caernarvon system, while being developed on Microsoft platforms, was not designed to operate in conjunction with any Microsoft product. Hence we decided it was necessary to change to a different CMS system. Ultimately, we used the Rational ClearCase and ClearQuest products. Since Rational is an IBM subsidiary, these products were freely available to us. They provide full integration of configuration management, bug reporting, and fault and change management.

Development of a system targeted at Common Criteria EAL7 requires extensive testing of the system, which in turn requires a comprehensive test suite that includes many hundreds or thousands of tests. This instantly leads to the conclusion that each test case must check the state returned from the system under test to determine if the expected result (i.e., success or a specific error or failure) has occurred. It is clearly impossible for a person to reliably check a large number of tests manually. The first test suite used was written to be run by an IBM internal tool, but it was not flexible enough for our needs. Our partners in Philips Semiconductors (now NXP Semiconductors) suggested changing to a test suite written in Ruby. This test suite has undergone extensive development, including facilities to exercise the internal functions of the system under test and to execute only some of the tests in a given run. In addition, a graphical interface program was written to make this simple to use. The program that executes the test cases has built-in facilities to compare and verify the results of each test. Thus, to attain the goal of all tests being self-checking, it is necessary only to ensure that, as each test is written, the expected return values are specified within the test. This slightly increases the time to write the tests but avoids much of the work that would be required if test verification code were to be written separately or added later.

The Common Criteria requires, at EAL7, that all code in the system is exercised by the tests. This is to check for “dead” code that can never be accessed. The toolset provided a facility to obtain details of all the instructions that were executed in a testing run but did not provide any means of matching this information back to the program source. A program was written to perform this match, for both C and assembler. This required yet another utility program to extract the debug information (in particular, the addresses of the functions and source lines) from the program binary file. The resultant categorization of source lines into executed, partially executed (e.g., clauses within an `if` statement), and not executed can be used as feedback to aid in the generation of additional test cases. While a primary goal of this code coverage measurement is to verify that there is no dead code, it should be noted that, on this processor (and probably others), certain instructions will always be inaccessible. The Smart*XA*2 processor requires that the target of branch instructions be on an even address—the compiler and assembler insert NOP instructions before the first byte of the destination of a branch wherever necessary to ensure the correct alignment. Such a NOP following an unconditional branch can never be executed. When submitting the system for EAL7 evaluation, it is necessary to explain these cases to the evaluators.

Due to the lack of test generation tools for the system, the tests were written by hand. If we were performing a development such as this again, we would attempt up front to obtain and modify a test generator (or write one from scratch if necessary) to automatically generate many of the tests for the system (see “Test Generation” below). Note that certain tests, in particular those for cryptographic algorithms, may not need to be written from scratch by the developer. In a number of cases the tests required can be obtained from sources such as NIST websites. However, these tests would require a program to convert the tests into a form that can be executed by the test system. Another problem with cryptographic tests is that, on slow processors such as smart cards, they can take a long time to run. For example, the full NIST test suite for the DES algorithm, including all of the Monte-Carlo type tests, could take months to execute on a smart card. In such cases, it would be appropriate for the developer and the evaluation agency to agree on a suitable subset of the tests to be run.

At EAL7, the Common Criteria required that the design of the Caernarvon system be built in layers; function calls could be made only to the same or lower layers—upward calls are not allowed. This layering requirement was specified by Schell [36], based on the work of Parnas [31] and Dijkstra [15]. The toolset in use did not produce an adequate cross-reference of the system that could be used to verify that this requirement was met. However, we had a program that produced highlighted listings of C and Assembler code and also generated a full cross-reference of the function calls in the system. A Ruby utility then processed this cross-reference file to flag any illegal upwards calls.

Static Analysis Tools

At EAL7, the Common Criteria requires that the developer perform a systematic vulnerability analysis of the Target of Evaluation. In an attempt to automate some aspects of the vulnerability analysis, we chose to apply static analysis of the implementation to find bugs that were not discovered by the extensive testing described above. Our search for a static analysis tool led us to BEAM (short for “Bugs, Errors, And Mistakes”), developed by another group at IBM Research [10]. BEAM met our requirements: it analyzed C language source code; the source to BEAM was available to add features required by our platform; and it cost nothing for internal use within IBM. Because BEAM ran only on UNIX, which was not one of our development platforms, we had to move a copy of the source code to a UNIX system for analysis by BEAM. For systems that have secrecy controls over the source code, having multiple (and complete) copies of the source code could present a problem. In our case, the source code was not considered secret, and its integrity was maintained elsewhere in the official repository.

As part of a related experimental program, we used BEAM to analyze the implementation of a component of our operating system, the file system, which was approximately 5,000 executable statements of C code. For this part of the project, we recruited other researchers to help us with the static analysis. They were experts with static analysis tools and understood operating system internals quite well, but were unfamiliar with the intricacies of smart cards. For this reason, we chose a relatively easy-to-understand, finite, well-defined body of code to start with. It was easy to analyze as a unit. Unfortunately, the project was suspended before we could expand our analysis to the rest of the C code in the system.

BEAM (with human assistance in interpreting the results) found about a dozen bugs and reported numerous violations of best practices. One common type of bug found was uninitialized variables; for example, an execution path could bypass the place where a variable was first set. Many of the data structures use short (8-bit or 16-bit) fields to conserve space, and BEAM found certain errors when performing arithmetic on these limited length fields, particularly if the operation was between a shorter and a longer variable. Our experience here indicates that, in any high-security project, it would be advisable to analyze all of the code of the system under development.

Other Tools We Desired

The development of any operating system is a lengthy process, and as a result, new technologies may arise during the course of the project that would have been very beneficial had they been available earlier. In this section, we will describe such new technologies as alternative approaches to the ones we used, and suggest possible fruitful research directions. However, as space does not allow us to do a comprehensive survey, we will limit ourselves to a broad overview.

One of the philosophies behind the Common Criteria is that no one technique or tool can guarantee a secure product, and as a result, multiple approaches should be used so as to reduce the chance of security flaws. Likewise, we do not claim that any particular tool or strategy (including the ones we used) is “best,” and we actively recommend the use of a wide variety of security-related tools. Broadly speaking, the tools can be divided into a number of categories based on their scope, such as life-cycle approaches, high-level design analysis, and test generation.

Correctness by Construction

Our work used a traditional software development methodology to meet the requirements of the Common Criteria. However, an alternate methodology, Correctness By Construction [14] has been advocated, and is relevant to this paper due to its reliance on specialized tooling support.

Correctness by Construction requires “using a sound, formal notation for all deliverables” and “using strong, tool-supported methods to validate each deliverable.” The process includes the early development stages, where even the user requirements have to be formalized to such an extent that it can be verified that the system design meets them. Programming is done in a language that incorporates formal mathematical assertions into the code, to enable verification of the code through static analysis. A system has been developed using this approach that meets criteria roughly equivalent to EAL7.

Even had Correctness by Construction and its related tools been available at the start of the Caernarvon project, it would not have been possible for us to make use of it. Since Caernarvon’s target platform was a smart card, we were subject to strong code size limits and the frequent need for low-level hardware access. The only available compiler was for C, and although subsetting this language to enable the addition of Correctness by Construction formal assertions has been attempted, this has failed. Retargeting an existing compiler for a more suitable language to our smartcard platform would not only be very labor intensive, but it is not clear that the result would have met our limitations. However, other operating system projects might not face these same obstacles, and so this methodology might be feasible for them.

Function Extraction

In contrast to previous work on generating correct code from specifications, the SEI CERT organization is conducting work on Function Extraction [20] [32], which aims to generate a complete behavioral description from examination of the code. This approach, similar to denotational semantics [38], treats a program as a mathematical function, mapping its inputs to its outputs. An extracted description can then be compared to the specification.

Currently ongoing is an effort to apply Function Extraction to Intel assembler language. Although this work is still in the early stages, it has the potential to detect security flaws introduced by a compiler and not found in the source code.

Test Generation

Common Criteria requires the application of a careful and systematic test methodology. Manually generated tests have historically been found to be poor, and therefore test generation methodology has been an active research area. Testing approaches are traditionally categorized as

- Black box - generating tests without knowledge of the system's implementation, or source code
- White box - generating tests derived in some way by analysis of the source code

"Grey box" or "glass box" testing is sometimes used to refer to testing that uses a limited amount of knowledge of the target system. The boundary between white and grey box testing is fuzzy, and we will not attempt here to draw a boundary between them.

An extensive survey of black box testing tools is found in [27]. In particular, fuzzing (originating from [28]) creates test inputs using controlled randomness. This technique has proven to be very effective in practice, as programmers often fail to defend against unanticipated inputs.

An example of white box testing is [11], where code is symbolically executed and conditions that cause different branching behavior are generated. These conditions are then used to generate test cases that exhibit these branches.

In general, there are a number of theoretical gains that can be made by enhancing strict black box testing with system information. Firstly, by monitoring the system under test, one can observe system failures that are not easily externally visible, and also determine whether some code portions have been undertested. Secondly, information extracted from the code or specifications can be used to target tests towards system portions that are more likely to contain problems. This has motivated work on grey box testing.

A recurrent issue in black box, white box, and grey box testing is internal state: most systems contain low-level state that is not evident in higher level interfaces. Examples of this include partitioning of file data into blocks, and page tables. Maintaining this internal state often introduces bugs. Since this behavior is not visible to a black box test generator, such generators often fail to detect these bugs. Although white box test engines do have access to the code, and therefore can make use of the low-level state management code, in practice this task is difficult because the system behavior depends not only upon the input parameters but upon the state established by all previous events. Inferring enough state properties to enable significant test improvement is extremely difficult for operating systems, as memory and storage subsystems contain much internal state. For example, although [11] has produced impressive results in essentially stateless systems, it fails to generate tests dependent on such things as the file system contents.

There are a number of approaches that have been developed to deal with state. Model-based techniques use a system model that describes the system states, providing information that can be used to verify proper behavior. A survey of model-based tools can be found in [33]. One state-based technique is described in the section below.

In our project, we did attempt to create a model-based test generation system. The intent was to create a test set based on the specification (i.e., model) of Caernarvon. As these tests were executing, monitoring would be done, and from this information further tests would be created to target code that the previous tests failed to execute.

As part of this effort, we developed a new taxonomy of security flaws [42], in order to be able to sensibly describe what flaws were and weren't subject to our approach. An important foundation question also had to be resolved: in order to test our system, we had to attempt to create certain states, but the APIs used to create those states were themselves under test. As described in our paper [30], we investigated this issue and found that in practice this was not an issue.

After designing, but before implementing our model-based test generation and feedback tool, we did an extensive experiment manually simulating its behavior using the Caernarvon file system as the target. In this experiment, we used the file system model to create tests. We executed the tests and analyzed the results. Unfortunately, as described in [41], this produced disappointing results, due to the fact that the internal file system code involved deeply complex flash memory operations, all of which resulting in enormous amounts of state that were not modeled by the file system API or directly related to any of the API operations. We therefore reluctantly concluded that the current model-based testing technology was not able to handle systems of the scale and complexity of Caernarvon.

State-Sensitive Testing

As with all developers of software that is to be Common Criteria certified at a very high level, we were required to perform extensive testing of our system. When we considered each of the hundreds of test scenarios we had written, we observed that each test scenario was composed of three distinct parts: pre-test setup, test execution, and post-test verification.

The pre-test setup involved state setup, i.e., the establishment of a particular environment in which we desired to run our test scenario. The task of repeatedly establishing the pre-test environment, even though it was automated through system calls, was very time-consuming. This was especially true when testing the cryptographic library, the file system, and the low-level persistent storage manager. A tool that could populate memory to establish the pre-test state without the need to actually use system calls would have saved us considerable time in the testing process. Unfortunately, such a tool for our smart card platform did not exist at that time.

Covert Channel Analysis Tools

In the past, a variety of covert channel analysis tools were developed for systems undergoing A1 evaluation under the TCSEC [7]. These were based on information flow analysis of the top level specifications and were part of the major formal specification languages in use at the time. These tools included a flow analysis tool for [17, 34] for SRI’s HDM formal specification system, Ina Flo [16] for SDC’s (later Unisys’) FDM formal specification system, and an information flow tool [24, 25] for the Gypsy language, developed at the University of Texas. Unfortunately, all of these tools suffered from excessive false positives [18, Appendix B].

Ideally, there would be a new generation of such tools that would work with more modern formal specification systems, such as PVS [29], ACL2 [23], or VSE [21]. Even better would be a tool that worked on source code, such as suggested in [19, 40].

Project Findings

Although our primary goal was to develop a high assurance operating system, we were forced to develop a considerable amount of tooling in support of our efforts. These tools helped us author, automatically generate, cross reference, and maintain a coherent, comprehensive set of documentation required by the Common Criteria. This documentation included (but was not limited to) the high-level design, the low-level design, and the test suite. Our tools also assisted us in automating portions of the testing and vulnerability analysis required by Common Criteria. The effort to develop new tools and extend existing ones was significant, and was measured in multiple person-years rather than in a few person-months.

In ordinary development projects, the development tools are usually available and work satisfactorily. In our high assurance project, however, the lack of suitable commercial off-the-shelf development and analysis tools in support of high assurance requirements significantly affected the process of development, prolonged the schedule, and increased the budget.

In the future, we recommend evaluating such tools early in the life of the project for their suitability to high assurance development. With enough early warning, project plans can be adjusted to include the development of missing or incomplete tools, if necessary.

Project Influence, Status, and Future

The Caernarvon project has had technical impact in the smart card industry, as well as in larger systems. The Caernarvon cryptographic library implementation was completed in 2003 and has been certified at EAL5+ in Germany [3]. This library was made available to vendors developing software for the Smart*XA*2 processor. The Caernarvon mandatory security policy was the basis for the Fuzzy Multi-Level Security Model [13] in the design of System S [43], a large-scale, distributed, stream processing system designed to analyze large amounts of unstructured data. The Caernarvon mandatory security policy was also used as a basis of the Simple Linux Integrity Module (SLIM) integrity policy in a Trusted Linux Client [35]. Lastly, the Caernarvon privacy-preserving authentication protocol [37] protects a smart card holder's identity and has been incorporated into the European CEN standard for digital signature applications on smart cards [1]. It is currently under revision by CEN [2] and will be submitted to the International Standards Organization (ISO).

An initial implementation of the Caernarvon operating system is fundamentally complete. It consists of approximately 33,000 executable statements written in C and 14,000 executable statements written in Assembler. The low-level design documentation is incorporated in the source code. In addition to the executable statements, there are approximately twice as many lines of comments. The software has been tested on a hardware emulator. If printed and stacked, the Ruby test source, the test framework, the test documentation, the operating system source, and the OS documentation would be several feet tall.

A working demonstration of an electronic visa application is complete. It demonstrates the use of Caernarvon access controls to permit authorized countries to read biometric data from and write entry/exit time stamps on each other's passports. It also demonstrates how the same access controls permit unauthorized countries to read public data but prevent them from writing anything. Lastly, it demonstrates how initialization data can be permanently write-protected from all countries.

The complete Caernarvon operating system has not been released as a commercial product. Although the project is currently inactive, future work will continue to address applications of this technology.

References

1. *Application Interface for smart cards used as Secure Signature Creation Devices - Part 1: Basic requirements*, CWA 14890-1, March 2004, Comité Européen de Normalisation (CEN): Brussels, Belgium. URL: ftp://ftp.cenorm.be/PUBLIC/CWAs/e-Europe/eSign/cwa14890-01-2004-Mar.pdf

2. *Application Interface for smart cards used as Secure Signature Creation Devices - Part 1: Basic requirements*, prEN 14890-1:2007, March 2007, Comité Européen de Normalisation (CEN): Brussels, Belgium.

3. *Certification Report for Tachograph Card Version 1.0 128/64 R1.0 from ORGA Kartensysteme GmbH*, BSI-DSZ-CC-0205-2003, 22 August 2003, Bundesamt für Sicherheit in der Informationstechnik: Bonn, Germany. URL: http://www.bsi.de/zertifiz/zert/reporte/0205a.pdf

4. *Common Criteria for Information Technology Security Evaluation - Part 1: Introduction and general model*, Version 3.1, Revision 1, CCMB-2006-09-001, September 2006. URL: http://www.commoncriteriaportal.org/public/files/CCPART1V3.1R1.pdf

5. *Common Criteria for Information Technology Security Evaluation - Part 2: Security Functional Requirements*, Version 3.1, Revision 1, CCMB-2006-09-002, September 2006. URL: http://www.commoncriteriaportal.org/public/files/CCPART2V3.1R1.pdf

6. *Common Criteria for Information Technology Security Evaluation - Part 3: Security Assurance Requirements*, Version 3.1, Revision 1, CCMB-2006-09-003, September 2006. URL: http://www.commoncriteriaportal.org/public/files/CCPART3V3.1R1.pdf

7. *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD, December 1985: Washington, DC. URL: http://csrc.nist.gov/publications/history/dod85.pdf

8. *Information Technology Security Evaluation Criteria (ITSEC)*, June 1991, Commission of the European Communities: Brussels, Belgium. URL: http://www.ssi.gouv.fr/site_documents/ITSEC/ITSEC-uk.pdf

9. Bormann, F.C., L. Manteau, A. Linke, J.C. Pailles, and J. van Dijk. *Concept for Trusted Personal Devices in a Mobile and Networked Environment*. in **Fifteenth IST Mobile & Wireless Communication Summit**. June 2006, Myconos, Greece. URL: http://www.orga-systems.com/file.php?mySID=%5Bvarout:mySID%5D&file=/research/IST-TPDConcept_v2.0.pdf&type=down

10. Brand, D. *A Software Falsifier*. in **11th International Symposium on Software Reliability Engineering**. 8-11 October 2000, San Jose, CA: IEEE. p. 174-185.

11. Cadar, C. and D. Engler. *Execution Generated Test Cases: How to Make Systems Code Crash Itself*. in **Proceedings of the 12th International SPIN Workshop on Model Checking of Software**. August 2005. URL: http://citeseer.ist.psu.edu/cadar05execution.html

12. Carter, M.G., S.B. Lipner, and P.A. Karger, *Protecting Data & Information: A Workshop in Computer & Data Security*, EY-AX00080-SM-001, 1982, Digital Equipment Corporation: Maynard, MA.

13. Cheng, P.-C., P. Rohatgi, C. Keser, P.A. Karger, G.M. Wagner, and A.S. Reninger. *Fuzzy Multi-Level Security: An Experiment on Quantified Risk-Adaptive Access Control: Extended Abstract*. in **IEEE Symposium on Security and Privacy**. 20-23 May 2007, Oakland, CA: IEEE. p. 222-227.

14. Croxford, M. and R. Chapman, *Correctness by Construction: A Manifesto for High-Integrity Software.* **Crosstalk**, December 2005. **18**(12): p. 5-8. URL: http://www.stsc.hill.af.mil/crosstalk/2005/12/0512CroxfordChapman.pdf

15. Dijkstra, E.W., *The Structure of the "THE"-Multiprogramming System.* **Comm. ACM**, May 1968. **11**(5): p. 341-346.

16. Eckmann, S.T. *Ina Flo: The FDM Flow Tool*. in **Proceedings of the 10th National Computer Security Conference**. 21-24 September 1987, Baltimore, MD: National Bureau of Standards and National Computer Security Center. p. 175-182.

17. Feiertag, R.J., *A Technique for Proving Specifications are Multilevel Secure*, CSL-109, January 1980, SRI International: Menlo Park, CA.

18. Gligor, V.D., *A Guide to Understanding Covert Channel Analysis of Trusted Systems*, NCSC-TG-030, Version 1, November 1993, National Computer Security Center: Fort George G. Meade, MD. URL: http://www.radium.ncsc.mil/tpep/library/rainbow/NCSC-TG-030.pdf

19. He, J. and V.D. Gligor. *Information-Flow Analysis for Covert-Channel Identification in Multilevel Secure Operating Systems*. in **Proceedings of the Computer Security Foundations Workshop III**. 12-14 June 1990, Franconia, NH: IEEE Computer Society. p. 139-148.

20. Hevner, A.R., R.C. Linger, R.W. Collins, M.G. Pleszkoch, S.J. Prowell, and G.H. Walton, *The Impact of Function Extraction Technology on Next-Generation Software Engineering*, CMU/SEI-2005-TR-015, July 2005, Carnegie Mellon Software Engineering Institute: Pittsburgh, PA. URL: http://www.sei.cmu.edu/publications/documents/05.reports/05tr015.html

21. Hutter, D., H. Mantel, G. Rock, W. Stephan, A. Wolpers, M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. *VSE: Controlling the Complexity in Formal Software Developments*. in **Applied Formal Methods - FM-Trends 98**. 7-9 October 1998, Boppard, Germany:Lecture Notes in Computer Science Vol. 1641. Springer. p. 351-358. URL: http://www.dfki.de/vse/papers/hmrs98.ps.gz

22. Karger, P.A. and H. Kurth. *Increased Information Flow Needs for High-Assurance Composite Evaluations*. in **Second IEEE International Information Assurance Workshop**. 8-9 April 2004, Charlotte, NC: IEEE Computer Society. p. 129-140.

23. Kaufmann, M. and J.S. Moore, *An Industrial Strength Theorem Prover for a Logic Based on Common Lisp.* **IEEE Transactions on Software Engineering**, April 1997. **23**(4): p. 203-213. URL: http://www.cs.utexas.edu/users/moore/publications/km97.pdf

24. McHugh, J. *An Information Flow Tool for Gypsy: An Extended Abstract Revisited*. in**!7th Annual Computer Security Applications Conference**. 10-14 December 2001, New Orleans, LA: IEEE Computer Society. p. 191-201. URL: http://www.acsac.org/2001/papers/140.pdf

25. McHugh, J. and D.I. Good. *An Information Flow Tool for Gypsy: Extended Abstract*. in **Proceedings of the 1985 Symposium on Security and Privacy**. April 1985, Oakland, CA: IEEE Computer Society. p. 46-48.

26. Mersey, D., *Caernarvon Castle - Fit for a Prince*, 2007. URL: http://www.castlewales.com/caernarf.html

27. Michael, C.C., *Black Box Security Testing Tools*, 28 December 2005, National Cyber Security Division, U.S. Department of Homeland Security. URL: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/black-box/261.html?branch=1&language=1

28. Miller, B.P., L. Fredriksen, and B. So, *An empirical study of the reliability of UNIX utilities.* **Communications of the ACM**, 1990. **33**(12): p. 32-44. URL: http://citeseer.ist.psu.edu/miller90empirical.html

29. Owre, S., J. Rushby, N. Shankar, and F. von Henke, *Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS.* **IEEE Transactions on Software Engineering**, Feb 1995. **21**(2): p. 107-125.

30. Paradkar, A., S. McIntosh, S. Weber, D. Toll, P. Karger, and M. Kaplan. *Chicken & Egg: Dependencies in Security Testing and Compliance with Common Criteria Evaluations*. in **IEEE International Symposium on Secure Software Engineering (ISSSE '06)**. 13-15 March 2006, Arlington, VA: IEEE Computer Society. p. 65-74.

31. Parnas, D.L., *On the Criteria to be Used in Decomposing Systems into Modules.* **Comm. ACM**, December 1972. **15**(12): p. 1053-1058.

32. Pleszkoch, M.G. and R.C. Linger. *Improving Network System Security with Function Extraction Technology for Automated Calculation of Program Behavior*. in **HICSS '04: Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9**. 5-8 January 2004, Waikoloa, HI: IEEE Computer Society Press. p. 90299c. URL: http://csdl.computer.org/comp/proceedings/hicss/2004/2056/09/205690299c.pdf

33. Redwine, S., *Introduction to Modeling Tools for Software Security*, 21 February 2007, National Cyber Security Division, U.S. Department of Homeland Security. URL: https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/modeling/698.html?branch=1&language=1

34. Rushby, J. *The Security Model of Enhanced HDM*. in **Proceedings of the 7th DoD/NBS Computer Security Conference**. 24-26 September 1984, Gaithersburg, MD: National Bureau of Standards. p. 120-136. URL: http://www.csl.sri.com/~rushby/papers/ncsc84-model.pdf

35. Safford, D. and M. Zohar, *Trusted computing and open source.* **Information Security Technical Report**, 2005. **10**(2): p. 74-82.

36. Schell, R.R., *A Security Kernel for a Multiprocessor Microcomputer.* **Computer**, July 1983. **16**(7): p. 47-53.

37. Scherzer, H., R. Canetti, P.A. Karger, H. Krawczyk, T. Rabin, and D.C. Toll. *Authenticating Mandatory Access Controls and Preserving Privacy for a High-Assurance Smart Card*. in **8th European Symposium on Research in Computer Security (ESORICS 2003)**. 13-15 October 2003, Gjøvik, Norway:Lecture Notes in Computer Science Vol. 2808. Springer Verlag. p. 181-200.

38. Stoy, J.E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. 1977, Cambridge, MA: MIT Press.

39. Toll, D.C., P.A. Karger, E.R. Palmer, S.K. McIntosh, and S. Weber, *The Caernarvon Secure Embedded Operating System.* **Operating Systems Review**, January 2008. **42**(1): p. 32-39.

40. Tsai, C.-R., V.D. Gligor, and C.S. Chandersekaran, *On the Identification of Covert Storage Channels in Secure Systems.* **IEEE Transactions on Software Engineering**, June 1990. **16**(6): p. 569-580.

41. Weber, S., P.A. Karger, S.K. Mcintosh, A. Paradkar, and D.C. Toll, *The Feasibility of Automated Feedback-Directed Test Generation*, RC 24355 (W0709-094), 24 September 2007, IBM Research Division, Thomas J. Watson Research Center: Yorktown Heights, NY. URL: http://domino.watson.ibm.com/library/CyberDig.nsf/Home

42. Weber, S., P.A. Karger, and A. Paradkar. *A Software Flaw Taxonomy: Aiming Tools At Security*. in **Proceedings of the 2005 Workshop on Software Engineering for Secure Systems—Building Trustworthy Applications**. 15-16 May 2005, St. Louis, MO: ACM. p. 1-7. URL: http://doi.acm.org/10.1145/1083200.1083209

43. Wu, K.-L., P.S. Yu, B. Gedik, K.W. Hildrum, C.C. Aggarwal, E. Bouillet, W. Fan, D.A. George, X. Gu, G. Luo, and H. Wang. *Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S*. in **Proceedings of the 33rd International Conference on Very Large Data Bases**. 23-27 September 2007, Vienna, Austria: ACM. p. 1185-1196. URL: http://www.vldb.org/conf/2007/papers/industrial/p1185-wu.pdf