RC24686 (W0811-036) November 5, 2008
Computer Science

IBM Research Report

Satisfying Common Criteria Security Evaluation Testing Requirements: Two Case Studies Using a High-Assurance Operating System

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

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Satisfying Common Criteria Security Evaluation Testing Requirements: Two Case Studies Using a High-Assurance Operating System

Matthew Kaplan, Paul A. Karger, Suzanne K. McIntosh
Elaine Palmer, Amitkumar Paradkar, David C. Toll and Sam Weber

Abstract—In this paper, we describe results of two case studies designed to satisfy the testing requirements mandated by higher levels of the Common Criteria for security evaluations. These requirements include 1) Demonstrable independence of individual test cases 2) Demonstrable coverage of both user level specifications and implementation code.
The case studies are based on a highly secure smart card operating system. In the first case study, we used automated specification based test generation, together with fault injection, to demonstrate that self-contained test cases are independent and do not lead to fault masking. The goal of the second case study was to demonstrate adequate code coverage, while retaining the ability to generate expected outputs based on specifications. To that end, we first established mappings between the specification and implementation code elements. Test cases were then generated from the user level specification to identify the executed code elements and we attempted to use static analysis to map the unexecuted code elements to the corresponding elements in the user level specification. In this second case study, we found that, given a sufficiently expressive user level specification, and a test generation system that is able to effectively use such a specification, the resulting tests will cover the vast majority of the code branches that are able to be covered. Therefore, the benefit of a feedback-directed system will be limited. We further provide evidence that the static analysis required to generate feedback in these cases tends to be difficult, involving inferring the semantics of the internal implementation of data structures. In particular, we observed that the internal states at the implementation level in a high security application pose significant challenges to this mapping process.

1 Introduction

This paper reports results of case studies attempting to satisfy requirements for testing in the upper assurance levels of the Common Criteria for Information Security Evaluation [1]. These testing requirements can be summarized as follows:

1) Ensure that testing is structured to avoid circular arguments about the correctness of the portions of the security functions being tested and document any dependencies among test procedures.
2) Ensure that each security function, each of its parameters (along with their boundary values), and negative testing based on high level specifications, are tested.
3) Ensure that each code element implementing the security function is tested completely.

The test dependency requirements have two consequences: 1) Need to develop *self-contained* test cases, where each test case in a test suite does all the set up it needs, verifies all the expected output including any system state updates, and cleans up after itself, and 2) Potential to minimize the risk of *fault masking*, where one fault prevents another fault from being exposed.

Our hypothesis was that a specification based test generation approach, such as the one reported in [2], would be able to help satisfy these requirements. This type of approach generates *self-contained* test cases that take into account boundary values, thus providing for generation of both positive and negative test cases for a given specification. However, the test cases generated using this technique may not achieve complete code coverage.

Recent works such as DART [3], CUTE [4], and EXE [5] have used code coverage to guide further generation of tests to ensure branch coverage. But these approaches generate only the test inputs that achieve the desired branch coverage and ignore the issue of generating expected test output (*test oracles*) — the ability to judge correctness of program output. We can employ the specification used to generate test cases to address this oracle issue. However, to achieve the necessary code coverage target using specification-guided test generation, unexecuted code must be associated with the relevant specifications - possibly leading to refinement of the specifications themselves. Although the possibility of such *round-tripping* has been mentioned previously [6], not only has this not been achieved, but, to our knowl-

• *The authors are all with the IBM Corporation, Thomas J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598, USA.*
E-mail: (mmk, skranjac, erpalmer, paradkar, toll)@us.ibm.com (karger, samweber)@watson.ibm.com

Manuscript received September 22, 2008; revised month day, year.

edge, it has never been attempted.

Our goal was to use the specification-based test generation technique reported in [2] in conjunction with code coverage information based on executed test cases. This test generation approach leverages the information provided in the UML class diagram and UML use cases of an application under test to produce a small yet effective set of *test objectives* and a corresponding set of *test cases*. This also addresses the issue of test oracle generation. We call this approach feedback-driven specification-based test generation.

As part of this work, we endeavored to demonstrate that round-tripping between system specification and concrete code was possible. We selected a significant real-world codebase as the system under test in order to provide future implementation efforts with guidance that is based on a non-trivial example, thus relating very directly to software generated for major development efforts. Our results indicate that mapping concrete implementation to user level specification is difficult to establish. The primary factor causing the difficulty was the state implicit in the implementation but not described in the specification.

1.1 Motivation

Analyzing software to look for security vulnerabilities is a challenging technical problem. Current practice relies on a combination of visual inspection, language-specific static analysis, detailed documentation, strict development processes, and extensive testing (by the developer and by the security evaluator).

Test case generation is particularly a problem for software that is intended to be evaluated under the Common Criteria [1]. The Common Criteria (ISO 15408) is a standard for the independent third-party evaluation of the security aspects of a computer system. For test case creation, the current state-of-the-art relies on manual technology, and even if done well, typically enables organizations to reach only Evaluation Assurance Level EAL4 or EAL5 (out of a possible 7) of the Common Criteria. This results in high costs due to long development cycles for the evaluated security components of systems, and lower target levels of security certifications. Tools that augment human analysis are required in order to meet a growing demand for evaluated products. This demand is driven primarily by recent mandates from the United States and Germany, and by financial institutions worldwide. One of the goals of our research has been to assist in meeting the high assurance test goals.

In the software engineering arena, the increasing use of UML (Unified Modeling Language) is making model-based automated test generation a new opportunity for significant productivity and quality enhancements. However, the current model-based automated test generation technology is not adaptive i.e. there is no feedback from the results of the test execution to guide further test generation. The motivation of this research was to dramatically advance the state of the art in automated testing, security analysis and certification. Our goals were (1) to employ high levels of adaptive test automation to facilitate the development of secure products and subsystems that can meet the rigorous testing requirements of Common Criteria EAL5 through EAL7 and (2) to apply the resulting technology to the world's first smart card implementation designed to meet EAL7. From the software engineering perspective, the new technology would offer heretofore unavailable predictors and metrics based on static analysis of source code and dynamic analysis of test coverage, as well as dynamic feedback into the test generation process based on these metrics. On the security side, these tools would improve the security of the target software, as well as facilitate gathering of test measurements required for third party security evaluations, which are mandated in order to sell security products to the US government.

1.2 Hypothesis

In our approach, a user level specification of the expected behavior of the application under test would be used to generate the test cases (along with the expected outputs for each test case). This approach is illustrated in Figure 1 below.

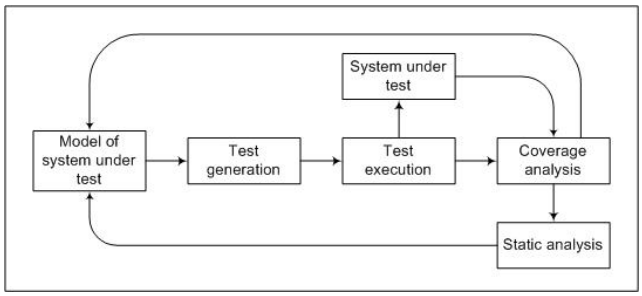

Fig. 1. Architecture Diagram for Feedback-Directed Automated Security Testing

Our hypothesis stated that it was possible, with the current state of technology, to design and build a specification-based test generation system that incorporates feedback to successively refine the tests it generates. More specifically, the system should

1) Take as input a user-level specification of the correct behavior of the system under test. The specification should contain knowledge at the level of normal API documentation and UML diagrams, but no implementation details.
2) Generate a set of *self-contained* specification-based tests that will cover all the expected behavior of the system. In cases where the specification states that the system's behavior depends upon its current state, the test cases will have to establish this state. Furthermore, unlike systems which generate random tests, the set of generated tests should be small. Such *self-contained* test cases do not lead to fault masking.

3) The test system should execute the current set of tests, monitoring which branches were taken by each test.
4) The code coverage information would be fed back to the specification-based test generator component. The test generator would use this information to either 1) generate a new set of test objectives and corresponding test cases to achieve the higher coverage, or 2) if necessary, refine the specification to account for the discrepancy with the implementation, and then generate new tests that, according to the revised specification, will likely cover branches that were omitted by the previous set of tests. The refined model need not correspond exactly to the concrete implementation details. It could be an approximate description, such as which parameters are relevant and irrelevant to a given code branch, to enable the system to tune its tests.
5) The process starting from step 3 will repeat until either full code coverage is achieved or until no further progress is made.

1.3 Organization of This Paper

The rest of this paper is organized as follows. Section 2 reviews some related prior work in feedback-directed test generation, specification-based test generation, and test dependencies. Section 3 describes our test subject and methodology, and Section 4 presents our results. These results are discussed and generalized in Section 5. Finally, we conclude in Section 6.

2 Related Work

The problem of demonstrating test independence is similar to the one encountered when integrating and testing object-oriented software, to decide the order of class integration. A number of papers have provided strategies and algorithms for deriving an integration and test order from dependencies among classes in the system class diagram [7], [8], [9], [10], [11]. The objective of all of these approaches is to minimize the number of test stubs to be produced, as this is perceived to be a major cost factor for integration testing. Indeed, stubs are pieces of software that have to be built in order to simulate parts of the software that are either not developed yet or have not yet been unit tested, but are needed to test classes that depend on them. Kung et al. [8] were the first researchers to address the class test order problem and they showed that, when no dependency cycles are present among classes, deriving an integration order is equivalent to performing a topological sorting of classes based on their dependency graph - a well known graph theory problem. In the presence of dependency cycles, the proposed strategy consists of identifying strongly connected components (SCCs) and removing associations until there is no cycle in the SCCs.

However, Kung and colleagues do not provide precise solutions when there is more than one candidate association for cycle breaking. In this case they simply perform a random selection. Existing solutions to this problem are based on the principle of *breaking* some dependencies to obtain acyclic dependencies between classes. Tai and Daniels [11] propose a 2-stage algorithm that deals with dependency cycles. However, in cases where class associations are not involved in cycles, their solution is sub-optimal in terms of the required number of test stubs. Le Traon et al. [10] propose an alternative strategy based on graph search algorithms that recognize strongly connected components, and that arguably yields more optimal results. One issue, though, is that this algorithm is not fully deterministic in the sense that, depending on some arbitrary decision (e.g., the initial vertex (class) of the search, and the search itself), the algorithm may yield significantly different results.

Furthermore, since the model used does not have any information on the kind of dependency (inheritance, association or aggregation), this approach may lead to the removal of an inheritance or aggregation relationship. Kung et al. [8], as well as others before them [11], point out that association relationships are usually the weakest links in a class diagram, i.e., they are the links involving the fewest dependencies and hence least complexity for stubbing.

This research is also the first attempt to incorporate feedback from test case executions into a process which incorporates test generation based on user level specifications. However, significant work has been reported in the areas of code-based and specification-based test generation, and here we present a brief overview of this work.

Godefroid et al. [3] present DART, a tool for finding combinations of input values and environment settings for C programs that trigger assertion failures and crashes when these programs are executed. DART combines random test generation with the use of a symbolic reasoning component for keeping track of path constraints that capture the outcome of executed control flow predicates. A constraint solver is used to determine from the recorded path constraints how subsequent executions can be directed towards uncovered branches. Experimental results indicate that DART is highly effective at finding large numbers of errors in several C applications and frameworks, including important and previously unknown security vulnerabilities.

The core approach combining concrete and symbolic executions has been extended to accomplish two primary goals:

1) To improve the scalability of the approach [12], [13] and
2) To improve execution coverage and bug detection capability through better support for pointers and arrays [4], [5], better search heuristics [14], or encompassing wider domains such as database applications [15].

The same goal of generating test cases to achieve branch coverage using genetic algorithms has been reported in [16], [17].

Yang et al. [18] used the EXE tool reported in [5] to detect errors in several file systems such as JFS. They used symbolic execution coupled with random data generated using constraint solving to detect crashes in the file system implementation. In contrast, our work uses user level specifications to identify deviations from the expected behavior.

Model-Based conformance Test Generation (MBTG) has become an area of active research [19], [20], [21], [2]. Typically, the models are described in terms of operations provided by the system under test (SUT) which maintains an internal state. Each operation is specified as a set of guarded results, and each result has a guard condition and a set of update actions on operation parameters and the system state.

Several MBTG techniques require the tester to additionally provide the test objective specifications to guide the test generation process [19], [20]. Such test objective specifications - independent from the behavior model - offer flexibility to the tester. Several techniques use an extended finite state machine (EFSM) as their fundamental representation for the models used in test generation [21]. EFSMs offer convenient adequacy criteria to guide the test generation such as State, Transition, and Predicate coverage. However, these techniques do not typically address the issue of generating a *verification sequence* - a sequence which enables verification of system state (both control and data) obtained as a result of applying each transition sequence. Petrenko *et. al* [22] describe a technique for generating such sequences for EFSM, however it is not applicable to UML object diagrams.

Kaplan *et al.* [2] uses UML use cases and class diagrams as the specification. Each use case flow has an associated guard condition and a set of updates (to the domain *object diagram* and the output parameters). Kaplan *et al.* then produce suitable *test objectives* which are refinements of the guard conditions on the use case flows using a set of fault models. This test generation approach also generates *verification sequences* as part of the *self-checking test oracle* to ensure that the object diagram updates associated with a given flow are implemented correctly. They use mutations on state updates to UML object diagrams to derive an appropriate distinguishing sequence to address the issue.

3 Case Study Description

This section describes the system used as a target for the case studies and the methodology used in conducting them.

3.1 Code Under Test

The software targeted for our case study was the file system component of the Caernarvon operating system [23].

Category	# of Files	Total Lines	Stmts	Branches	# of Func
C Header	73	16324	4994	0	0
Non-FS C	114	65038	21170	3064	393
FS C	24	15943	4778	1080	101
Total C Files	211	97305	30942	4144	494
Assembler	31	26961	8682	1049	413
Total	242	124266	39624	5193	907

TABLE 1
Caernarvon Code Metrics

We chose Caernarvon on the grounds that it is well defined (there is a complete and accurate specification), and it is of a suitable size (large enough to give meaningful results, but not so large as to be unwieldy). Caernarvon is a high assurance operating system that was under development at IBM T.J. Watson Research Center for use in embedded solutions such as smart card applications. Our target hardware platform was a smart card chip, and operating systems designed to run on such devices must be optimized to use very little memory because a typical smart card chip might only have 8K bytes of RAM and 128K bytes of persistent storage.

Because the Caernarvon operating system is intended to pass the highest assurance level (EAL7) of the Common Criteria [1], there is a very strong requirement to keep the code as simple as possible. To assist in reducing complexity, Caernarvon is designed as a strictly layered operating system, with upward calls forbidden.

Table 1 shows the size of the entire Caernarvon operating system, with the file system layer broken out, measured in total lines of code (including comments and white space), numbers of C and assembler statements, and the size of header files (declarations). The File System is written entirely in C. The majority of the lines of assembler code are found in the cryptographic library.

At a high level of abstraction, the Caernarvon operating system provides a well-defined set of services, implemented as supervisor calls. These services are grouped into logical subsets such as the cryptographic services, the key management services, the file system, etc. For the purpose of this paper, we considered only the file system services, and modeled the behavior of each file system service as a UML use case. The Caernarvon operating system provides a variety of security features for the file system, including mandatory and discretionary access controls and storage quota controls. The file system also provides facilities to map files into virtual memory – this feature was excluded from our model.

Externally, the Caernarvon file system looks much like a conventional file system, with typical file create, open, read, write, etc calls. Internally, though, the implementation is very different, due to the extremely limited memory space on a smart card. The Caernarvon file system does not actually store directories at all. Instead, each file contains a pointer to its parent directory. Path name translation is carried out by searching all of memory for

the file whose name and parent name matches the path name. This saves a significant amount of space, while the performance cost is quite small, because there is not very much memory through which the system has to search. The implication of this for testing is that the test cases generated based on system specification may not be able to cover the entire low-level implementation.

The Caernarvon file system operations were complicated by the need to consider the relative priority of errors. If an operation encounters multiple errors, returning the wrong one could leak information, causing a security problem. For example, when a file creation operation is attempted, the fact that the file already exists must not be divulged unless the user has permission to read the directory.

3.2 Methodology

Briefly, our methodology was to go through all computations that an idealized feedback-directed test system would entail, doing hand-simulations wherever necessary. In detail, the steps were:

1) Creating UML models of the external interfaces to the file system (specifically, parameters, success results, and error conditions of the top level supervisor calls), as well as modeling the system's state and semantically-significant object classes.
2) Generating test objectives, which formally express tests for specific fault sensitivity and are intended to identify particular faults (e.g. attempt to write to a file open only for read).
3) Generating tests that fulfill the test objectives. These are ultimately expressed in the actual test execution language (in our case, Ruby).
4) Grouping the tests into categories (e.g. all tests of "file write").
5) Running the tests, capturing data on which C blocks were executed, partially executed (because of multi-part conditionals), and not executed. This activity includes:
   - Mapping the executed machine language instructions back to the C source using the output of the development environment.
   - Capturing data on which lines were unique to each test, which lines were common to multiple tests, and which lines were not executed by any tests.
   - Excluding lines that are part of pre-testing setup and post-testing verification, and not part of the lines of interest. For example, in order to open a file, it must be created first. The lines relevant to creating a file must be excluded when testing file opening.
6) Analyzing the C source code to determine why the unexecuted lines were not executed and identifying what additional information is required as feedback to the test generation system, so as to force the execution of the unexecuted lines. This step entailed the following:
   - analyzing the branch condition that enveloped the unexecuted lines.
   - analyzing inter-procedural control and data flow.
   - establish or reject the existence of a linkage from the immediate branch condition through outer branch conditions, all the way to top level operation parameters and/or global system state.
   - locating all transformations on that top level operation parameter and/or global system state, including inter-procedural transformations, that would influence the content of the feedback 'message'.

Some of the steps above were performed automatically, some manually, and some were simulated. Specifically, step 1 is a task typically done by a human expert, and would continue to be done this way in the final system. Steps 2, 3, and 4 would normally be done by an automated test generation system, but were simulated using manually-written and hand-selected tests. Step 5 was automated with custom-written tools that analyzed the raw data from our unusual development environment (the smart card emulator). Step 6 was performed manually.

The intention of this case study was to investigate the feasibility of automating step 6. Steps 1 through 5 may provide valuable feedback information to a human expert, but it is step 6 which would be the focus of automated feedback. In our second case study, we used the same target environment and test cases to investigate the presence of fault masking, which can be observed independent of the presence or absence of any type of feedback mechanism.

3.3 Modeling Details

As described above, the first step in our experiment was to create a model of the Caernarvon file system based upon the Caernarvon specification. This would be given to our simulated specification-based test generation tool. Our model was UML-based, with augmentations to enable expression of the full Caernarvon file system semantics. As such, it was at least as expressive as the models used by any specification-based testing tools that we are aware of. As it was able to fully describe the Caernarvon file system, any failure of our hypothetical system would not be due to limitations of our modeling language.

A UML domain model described the semantic entities in the Caernarvon file system, as shown in Figure 2. It consists of 5 classes. Classes `File`, and `Directory` inherit from an *abstract* class `FileSystemObject`. A `Directory` may contain other `FileSystemObjects` (and hence both other `Directory` and `File` instances). This is depicted by the parent-children association

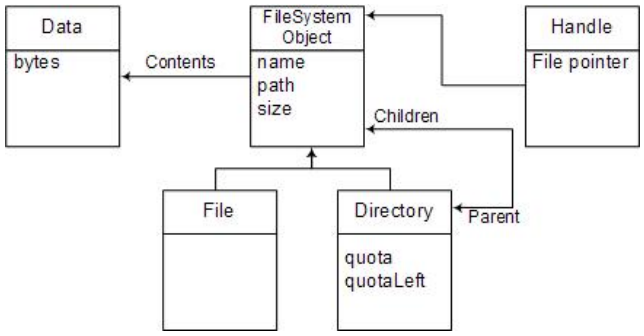

Fig. 2. Domain model of Caernarvon FS

between the two classes. A special instance of the `Directory` class - called `root` - does not have a parent. A `FileSystemObject` consists of blocks of `Data` (represented as a *Sequence*). Also, a `FileSystemObject` may be referred to by a `Handle` object. Each `Directory` instance has a `quota` attribute to indicate its allocated maximum amount of storage space and a `quotaLeft` attribute to indicate the remaining available space.

The other part of our model described the file system operations, including their effect upon the system state. UML use case diagrams were greatly extended by a mini-language describing the conditions under which every distinct behavior of a given file system operation occurred, and the state changes that result from said operation. A parser for this mini-language was produced to assist us. The model describing the read operation is shown in Figure 3. As can be seen, steps 1 and 2 describe the normal, successful, operation of the read operation. `e1.1` through `e1.4` are the exceptions that can be raised after step 1. Each exception has a brief English language description of the exception's cause, a predicate stating under what conditions the exception will occur, and finally a description of what computation the exception processing will perform. For example, `e1.1` describes what happens when a read past the end of file is attempted. In this case, the computation described in step `e.1.1.1` occurs, which sets the return code appropriately, and outputs the remainder of the file's data and the number of bytes read. In this example, each exception performs only one step of computation, but in general many steps can occur, and other exceptions raised during exception processing.

As can be seen, our use case specifications were computationally complex.

3.4 Test Generation and Execution Details

After completion of the file system model, we generated tests based upon it. Test generation proceeded in two stages simulating the techniques reported in [2]. First, sets of *test objectives* were created. Each test objective was described in terms of the model entities, a system state, and a test operation to be performed when the system is in that state. For example, one test objective would be

```
Use Case FS_Read:
  usecase Caernarvon file system read
  precondition none
 1. Ruby inputs file handle, byte count
    InputParameters:
      Declaration fileHandle: FileHandle
      Declaration byteCount: Integer
 2. System outputs requested number of bytes & QSVCgood
    OutputParameters:
      Declaration readData: Integer
          Value: readData:=SUBSEQUENCE(
            fileHandle.file.data,fileHandle.filePointer,
            fileHandle.filePointer+byteCount)
      Declaration returnCode: ReturnCode
          Value: returnCode:=ReturnCode.QSVCgood
 e1.1.  exception the data in the file isn't long enough
          to read count bytes starting with file pointer
      Predicate: fileHandle.filePointer+byteCount>
            fileHandle.file.data.COUNT()
 e1.1.1. System returns partial data, count of bytes
            read, and QSVCoutside
    OutputParameters:
      Declaration readData: Integer
          Value: readData:=SUBSEQUENCE(
             fileHandle.file.data,fileHandle.filePointer,
             fileHandle.file.data.COUNT())
      Declaration returnCode: ReturnCode
          Value: returnCode:=ReturnCode.QSVCoutside
      Declaration bytesRead: Integer
          Value: bytesRead:=fileHandle.file.data.COUNT()
              -fileHandle.filePointer
 e1.2.  exception the file referred to by file handle
             has been deleted
      Predicate: fileHandle.file.deleted
 e1.2.1. System returns QSVCdeleted
    OutputParameters:
      Declaration returnCode: ReturnCode
          Value: returnCode:=ReturnCode.QSVCdeleted
 e1.3.  exception the file handle is not valid
      Predicate: fileHandle=null
 e1.3.1. System returns QSVChandle
    OutputParameters:
      Declaration returnCode: ReturnCode
          Value: returnCode:=ReturnCode.QSVChandle
 e1.4.  exception arithmetic overflow on buffer position
      Predicate: fileHandle.filePointer+byteCount>
              theFileSystem.mAXINT
 e1.4.1. System returns QSVCbadparm
    OutputParameters:
      Declaration returnCode: ReturnCode
           Value: returnCode:=ReturnCode.QSVCbadparm
```

Fig. 3. Use case Model for `Read` Operation

to invoke the file open operation on a file that doesn't exist. Secondly, for each test objective, an actual test was created that performed the operations necessary to create the desired initial state, and then executed the operation under test.

Test objectives capture information about the state and set of parameter values necessary to trigger desired behavior. In order to test each operation's error conditions, we generated a test objective for each exception, consisting of the exception's predicate. If a disjunction occurred in the exception's predicate, then the exception was split into two, and objectives generated for each subpredicate. Objectives to test the normal execution were generated by conjoining the negation of each of the exception predicates, and splitting the resulting predicate at each disjunction. The domain model also implies "interesting" system states which we want to be tested. For example, given a recursive model structure, one wants to test the system in states where the structure recurses zero, one, and some other positive number of times. The

special states so generated are composed with all of the objectives created by examination of the exceptions. That is, if there are three special states identified, then we should try to test that each exception will occur in each of those states.

Some objectives resulting from the above algorithm are self-contradictory. For example, since the state with no files is a special state, and attempting to read more data than a file contains is an exception, we would have an impossible test objective of attempting to read more data than a file contains when there are no files in the system. These objectives were discarded. For the remaining objectives, a sequence of operations was created and a set of parameter values selected so that the test objective was satisfied. In cases where certain values were under-specified (such as the name of a file to be read), arbitrary values were chosen. It should be observed that creating a sequence of operations which will produce a desired system state is a non-trivial problem. In general this problem is undecidable for any Turing-complete specification language. Nevertheless, we assume that our idealized test generation tool uses planning algorithms which can solve such problems under common circumstances.

In order to be able to generate a set of *self-contained* test cases, We define 3 kinds of dependencies between a pair of operations:

1) **Set Up**, which arises when an operation needs to be invoked with a successful result before another operation can succeed,
2) **Verification**, which arises when an operation needs to be invoked in order to verify behavior of another operation, and
3) **Clean Up**, which arises when an operation needs to be invoked to reverse the state update effects of another operation.

We derive these dependency relationships among a set of operations, and use a two-phase topological sort of these dependencies to derive the necessary self-contained test cases

Each test was executed on the test system, with monitoring done to determine which instructions were executed. Instructions that were executed in order to create the proper initial state for a test were disregarded. This information was then analyzed to determine which lines of the C source code were not executed.

4 Case Study Results

4.1 Fault Masking Case Study Analysis

In support of our experiments, we injected known faults into the stable implementation of Caernarvon available to us by modifying the source code. Fault injection is a commonly employed technique to validate hypotheses in testing experiments.

We performed testing of the Caernarvon Builds with Injected Faults (BIFs) by running a standard set of test cases (267 of them) against each build. These test cases target all Caernarvon operations, *i.e.* they are not limited to testing the File System services only.

For every build, we required that all standard test cases must have been attempted and a clear *pass* or *fail* indication received. These criteria ensured that both the test environment and the SUT remained healthy in areas outside the scope of the injected fault. We discarded the BIFs (and the associated faults) that prevented the full suite of test cases from completing.

Another utility, the Caernarvon Fault Injection Tool (XFIT), augments the basic Caernarvon test environment already described. In order to facilitate the anticipated high volume of testing, XFIT supports:

1) Injection of single faults and fault combinations into Caernarvon builds;
2) Automatic generation of Caernarvon BIFs;
3) Automatic testing of BIFs;
4) Automatic storage of test results for analysis.

XFIT provides a simple external interface to facilitate the fault injection process. A user/tester may select any combination of faults and, through a menu-driven interface, one can initiate the automatic generation and test of Caernarvon BIFs. XFIT collects test results, unambiguously associating the results with the faults injected.

In preparing for our fault injection experiments, our first task was to identify the types of faults to be injected into the implementation of Caernarvon File System services. An initial list of *plausible* file system bugs was produced. To avoid bias, this list was produced from the file system services specification document alone by one of the authors who was not familiar with either the implementation details or the existing Caernarvon test suite.

Of this list, fifty faults were selected and implemented. Although all of them could have been implemented, we preferred faults that:

1) Required minimal changes to the existing code, and
2) Could be fully realized without the need to modify multiple software components.

We injected each of the fifty faults individually, generating fifty BIFs. We quickly found that some of the faults we had injected interfered with Caernarvon's initialization sequence - which creates, among other things, the root directory of the file system. To combat this, we introduced a flag for gating the injected faults such that the injected faults took effect only after Caernarvon entered its steady state.

Gating in place, we proceeded to run the full suite of test cases against this collection of BIFs. We discovered that fourteen builds either caused catastrophic failures or failures that occurred early in the setup stage of a test case. In both of these scenarios, we found that only a portion of the standard test suite had run. As stated earlier, we required that all test cases run and yield a clear *pass* or *fail* indication. Consequently, these fourteen BIFs (and the associated injected faults) were removed from our target set.

Our final fault set consisted of thirty six faults quite representative of common programming errors especially prevalent in file processing systems. For example, failure to:

- verify path or filename is legal
- charge or return quota properly
- create file of correct size or file type
- prevent operations illegal for a given file type or mode
- record file state information accurately
- initialize/update/use the file pointer properly
- trap a read beyond the end of file

In Table 2, we illustrate some of the injected faults. The column labeled **Operational Code - No Fault** shows the fragment of correct code, and column labeled **Code with Injected Fault** shows the code corresponding to the injected fault. For example, `Fault_StartRdAtWrongAddr` in Table 2 causes an off-by-one error with respect to the file pointer during a file read operation. We inserted code that directly decrements the file pointer prior to calling the low-level read operation. Other faulty code shown in Table 2 causes side effects such as:

- Reducing the correct file size prior to creating the file.
- Neglecting to update quota (achieved by commenting out certain code).
- Returning the wrong amount of quota.

Having identified our target set of single faults and scrutinized the single fault seeding results, we turned our attention to experimenting with paired fault seeding. We paired inter- and intra-service faults and then ran the standard 267 test cases. When we refer to *paired inter-service* faults, we are referring to two faults, each of which occurs in implementations of different services. When we refer to *paired intra-service* faults, we are referring to two faults, both of which occur in the implementation of the same service.

Once again, we scrutinized the results and discarded fault pairs that did not meet our criteria - namely, all test cases must run and all test cases must give a clear *pass* or *fail* indication. Our paired fault set consisted of 207 paired fault BIFs (135 inter–service, 72 intra–service). Together with the results of single fault testing, we would use the output of these tests to search for signs of fault masking.

We compared the results obtained for each paired fault tested, to the results obtained when each fault comprising the pair was individually tested. To claim absence of fault masking, we needed to demonstrate that the union of test cases that failed for each of the two faults tested individually was present in the results obtained when the two faults were paired up and tested.

The result of this analysis provided us with several candidate cases of fault masking - it appeared we could not claim absence of fault masking. Upon further investigation, we were able to characterize these candidate cases as belonging to one of two categories of masking which were actually trivial, the direct result of the presence of opposing faults. The two categories are:

1) Masking that occurs from the presence of two faults that cancel each other out.
2) Masking that occurs from the presence of two faults that cannot co-exist logically.

As an example of the first category, consider enabling two faults: `Fault_Add1` and `Fault_Sub1`. Enabling `Fault_Add1` demands that a pointer be incremented by one. Enabling `Fault_Sub1` demands that same pointer be decremented by one. If we include both faults, the net result is an unchanged pointer. We noted five such cases. Not surprisingly, we observed cases such as this only with intra-service fault pairs where the fault pairs are likely to modify code within the same module.

As an example of the second category, consider enabling two faults: `Fault_Wr2RdOnlyHandle` and `Fault_Wr2ClosedHandle`. Enabling `Fault_Wr2RdOnlyHandle` requires that the handle to be written to be open in read-only mode. Enabling `Fault_Wr2ClosedHandle` requires that the handle to be written to be closed. The prerequisite state of the file handle cannot be both open and closed. Therefore, this is an illogical combination. We noted nine such cases.

We also observed other interesting results not associated with fault masking. In these results, we discovered that one fault pair caused test cases to fail even though these test cases had passed when we applied each of the two faults of the pair individually. Six test cases failed in this instance. One of the test cases, a File Seek test, failed on file open because we reached the maximum number of open handles and, due to one of the faults injected into the build, we prevented file handles from being marked as closed. The other five test cases failed because they attempt to create a file that already exists. Normally, the failed File Seek test, which had created the same file, would have deleted it. However, owing to its own failures, the File Seek test had not deleted that test file. This implies that for a test case to be truly *self-contained* it is not enough to have all clean up fragment in the test cases, its placement within the test case is also important.

4.2 Feedback Case Study Analysis

The most critical component of this experiment was the process of determining, for every unexecuted code branch, whether or not it could be generated via a reasonably feasible feedback. To accomplish this, we started with two basic analysis techniques. First, for every code branch that leads to an error condition, we would do *static slicing* to determine which structures influence the test condition. These slices would be compared, to isolate the implementations of model elements. This information would be augmented by information inferred by the code's loops. Secondly, we would do data flow analysis, determining which data structures are

Id	Operational Code - No Fault	Code With Injected Fault
Fault_StartRdAtWrongAddr	rc = PSM_ReadObject (DataMemId, Offset+FcbEntry->Position, CntData, (USHORT_P)Buffer)	rc = PSM_ReadObject (DataMemId, (Offset+FcbEntry->Position)**-1**, CntData, (USHORT_P)Buffer)
Fault_CreFileWrongSize	rc = PSM_CreateObject (MS_FILE | MemType, HdrSize + FileSize, &HdrId)	rc = PSM_CreateObject (MS_FILE | MemType, (HdrSize + FileSize)**-1**, &HdrId)
Fault_QuotaNotCharged	RelevantQuoteMemId = ParentMemId; AvQuota = Quota; rc = fsManagemQuota(&RelevantQuoataMemId, &AvQuota, QT_SUB)	RelevantQuoteMemId = ParentMemId; AvQuota = Quota; **/*** rc = fsManagemQuota(&RelevantQuoataMemId, &AvQuota, QT_SUB)***/**
Fault_WrongQuotaCharged	AvQuota = HdrExt.qc.Quota;	AvQuota = HdrExt.qc.Quota **- 1**;

TABLE 2
Faults Injected into File System Implementation

influenced by each of the parameters to the service call. As we know the mapping between service parameters and model elements, we could therefore detect which data structures pertained to each model element. Further, for those code branches that could not be reasonably covered via feedback, we had to decide whether or not such a failure should be considered evidence against our hypothesis.

The latter point deserves careful consideration. Although at first glance it might appear that all uncoverable code, i.e. code that was not exercised by a given test, represents a rejection of feedback-directed test generation, this would be too high a bar to be realistic. We argue that there are two common cases of legitimate uncovered code: *low-level semantics*, and *modeling boundaries*, as well as one rare case: *impossible* cases.

In almost any real system, there are conditions which are of too low a level to be feasible or useful to be represented in a system model. For example, most programs are subject to out-of-memory errors. To completely model such an error, i.e. be able to determine exactly when the system will generate a memory error, would require not only a specification of the size of all of the program's data structures, but details about such things as the malloc/free block consolidation policies. Not only would such a detailed specification almost certainly be economically infeasible, it would also be undesirable on the grounds that such a specification would tie the program to a particular system and compiler version. In our test system, code to deal with both out-of-memory conditions and hardware failures fell into this category. The code fragment below demonstrates this case, where line 463 is not executed in any of our tests since this requires injection of EEPROM errors.

```
...
458 rc=PSM_ReadObject(MemId,//PSM_HANDLE handle
459 sizeof(IFINFO),//offset into memory object
460 HdrExtSize, // length
461 (USHORT_P)&TempExt);// output buffer
462 if (rc !=QSVCgood)
463 return(rc);
...
```

In any modeling endeavor, decisions must be made about where to draw the line between system aspects that should be modeled and those that should not. Very rarely is an entire system modeled, but rather coherent subsets. Our test case was specifically limited to the file system, for example, rather than the entire OS. Further, the file system had operations for memory-mapping files, and this functionality was omitted because the successful modeling of these operations would require a very low-level model. Code to deal with such excluded functionality does not need to be covered. As a result, we had to determine whether any uncovered code was the result of any omitted functionality.

A probably rare occurrence is what we call "impossible" conditions. Consider, for example, password checking code. Commonly, such code performs a cryptographically-secure hash on the input and compares it to the stored hash of the correct password. An automatic test generation program, in order to test the "password correct" execution path, would have to break the secure hash algorithm and thereby determine the correct password. Naturally, this is computationally impossible.

Before we obtained any experimental data, we planned algorithms for generating feedback. Although we were open to any techniques that would become apparent upon viewing the experimental data, the prior planning enabled us to not only attack the data systematically, but also discuss what inferences are infeasible, without being biased by the actual data.

We observed that error conditions provided significant information about the program structure. An error condition that corresponds to a certain state element being improper has to be implemented via accesses to that element. Therefore, the test conditions leading to that code branch, and the computations in the code path leading to that test which affect the test's outcome must include the data structures that implement that model element. Further, by comparing the code branches between different error conditions, one can refine our inferences about the structure meanings.

We also expected that recursive model elements would be implemented via loop or recursive structures. Therefore, we could attempt to infer the implementations of such structures by examining the conditions being looped over.

4.3 Feasibility of Mapping

The process of generating test objectives and the tests to meet them resulted in 63 objectives and an equal number of tests, as shown in *Objectives* column in Table 3. The number of objectives for each operation can be considered to be a measure of the complexity of its specification. For example, Create could produce many exceptions (file already exists, out of quota, no permission, and so forth), while Close had few.

The execution of our tests resulted in a total of 142 uncovered or partially covered code blocks. Table 3 contains a breakdown of them by operation.

Automatic refinement of the model to generate additional test cases is what we define as the opportunity for automated feedback. As can be seen in Table 3, only 19 of them, or 13%, were classified as opportunity for automated feedback. The non-opportunity cases represented those which were too low-level to be feasible, places which were beyond our modelling boundaries, and impossible conditions. The last did not occur in our case study. The relatively small opportunity was unexpected, and for some purposes this alone would indicate that feedback-directed test generation was not sufficiently profitable.

As we progressed through our code inspection, we found it revealing to sort the lines representing opportunity into three sub-categories: cases that necessitated creating the proper system state, cases that involved obtaining certain parameter values, and those cases that involved both state and parameters.

The one parameters-only case consisted of a test for a special case, which simply compared one of the function parameters to a constant. The code fragment below demonstrates this case, where line 224 is not executed since there is no test for `ip->PathLen == 0`.

```
...
223 if (ip->PathLen == 0)
224   return(RC_CannotDeleteMF);
...
```

Here, `ip->Pathlen` is a field in the structure passed as an argument to the top level operation invoked from the external environment. If the model corresponding to this top level operation also had an input parameter that represented the same information, the necessary mapping to the model parameter would be straightforward. In this case, the model would not need to be refined to generate the extra test case. Instead, a new test objective corresponding to the predicate would be sufficient to generate the necessary additional test case to cover the branch.

The cases that involved state, however, turned out to be problematic. One of the actual examples considered occurred when testing the Delete operation, whose purpose is to delete files. This operation starts executing the function `FileDelete`, which, in turn, can call the function `fsDeleteFile`. A branch in this latter function isn't taken. This code is shown below:

```
startidx=0; while (1) {
  FCBEntry = fsSeekMemIdinFCB(MemID,
             startidx, PROG_NO_PID);
  if (FcbEntry == 0)
    break;
    ... unexecuted lines ...
 }
```

In order to understand how to cover these unexecuted lines, either a human or an automated system has to determine that:

1) The `MemId` parameter to `fsSeekMemIdinFCB` is an internal representation for the file whose name is passed into the Delete operation,
2) Because the third parameter to `fsSeekMemIdinFCB` is `PROG_NO_PID`, this while loop will cause `fsSeekMemIdinFCB` to examine all file handles held by all users, and
3) `fsSeekMemIdinFCB` will only return a non-zero result if a file handle happens to be both open and referring to the file represented by `MemId`.

In fact, these lines can be executed only if someone (not necessarily the caller) has opened the file to be deleted. Clearly, this requires detailed understanding of the internal representation of file handles. The likelihood of automatically deciphering this is made even more remote when we observe that the specification of the Delete operation does not even mention file handles or the Open SVC – this code is not actually necessary to implement the delete functionality, but exists to optimize the read and write SVCs.

Space considerations prevent us from displaying more system code, and therefore we will content ourselves with higher-level discussion about feedback generation obstacles that we discovered.

The Caernarvon filesystem was designed to store its data in relatively small EEPROM or flash memories. Filesystem operations tend to fragment memory, meaning that filesystem data has to be relocated in a garbage-collector-like fashion. As a result, each file object has an identifier, and references to other objects are made by these identifiers, not by pointers. In order to locate an object via its identifier, the system performs a linear scan of memory until it finds the correct entry.

Additional complications arise with the Create operation (and some others), as a result of the quota functionality. A check that there is sufficient quota must be performed before every file creation. Our model, like most filesystem implementations, describes this as comparing the requested quota to a "quotaLeft" field. However, in order to save space, the Caernarvon filesystem does not store this value but instead computes it by walking through the subtree, counting the quota actually used, and subtracting it from the quota limit. Such implementation departures from a user-level model

			no opportunity			opportunity		
	Objectives	uncovered blocks	low-level	model boundary	impossible	parameters	state	both
Create	17	50	21	27	0	0	0	2
Open	16	27	10	12	0	0	0	5
Read	8	13	7	4	0	0	1	1
Write	7	15	7	4	0	0	0	4
Tell	7	2	1	1	0	0	0	0
Close	3	0	0	0	0	0	0	0
Delete	5	35	10	19	0	1	0	5
Totals	63	142	56	67	0	1	1	17

TABLE 3
Coverage Statistics

are not only common in general, but actually usually considered desirable as they provide an independent means to verify an implementation's correctness.

We argue that these obstacles make the use of static analysis to produce user-model refinement infeasible at the current state of technology. Shape Analysis (as described in [24] and similar work) is an active research area that attempts to determine whether code that uses heap-allocated memory correctly implements higher data structures such as linked lists and trees. This body of work provides an illuminating comparison to our mapping problem.

Currently, Shape Analysis is at the level of determining whether list and tree implementations that use pointers agree with a given low-level specification. In our case study, we have to infer meaning from code in which references are not pointers but identifiers, deducing that memory scans are equivalent to dereferences, and overcome the fact that the filesystem tree implementation does not represent connections between a directory and its content. Furthermore, our model describes only a high-level API, not the low-level implementation functions. Even though it would suffice for feedback to give only approximate answers, it is still the case that generating feedback in our case study would be considerably harder than Shape Analysis. Since doing Shape Analysis is itself considered a difficult research problem, we conclude that our case study code is not amenable for feedback-directed testing.

5 Discussion

In this section, we discuss the negative results obtained in the feedback case study. Prechelt [25] has argued that too few negative results are published in this field, and that this has resulted in losing insights that such work provides. We observe that in previous publications, round-tripping has been proposed and sometimes assumed to be trivial. The results of our case study show that this problem is not only non-trivial, but also has serious barriers to success. In this section we discuss how generalizable these results are.

While our results may not be applicable to other situations, we do *not* believe that the results of this case study are generalizable to other filesystems. Most other filesystems have APIs that specify more low-level behavior (like inodes), and have implementations that are much closer to the specification.

The primary cause of our reverse-mapping difficulties was the use of state. We distinguish between two classes of state: explicit and implicit. Explicit state is that which is articulated in the system specification and included in the system model. In our test case, this includes the directory structure, the file names, and so forth. Implicit state is that which exists in the implementation but which is at too low a level to be described in the specification. In our test case, this includes how the file data is divided into blocks of memory. In other applications this would include such things as caches and internal id numbers.

Explicit state is clearly a more tractable venue for generating feedback than implicit state. This is true because in the case of explicit state, one has available both the specification of an entity and the code that directly implements that entity. Additionally, we do not need to prove a correspondence between the code and the specification, we merely need heuristics that are likely to fruitfully guide further test generation. This is a closer match to Shape Analysis. In our case study, however, the uncovered code involved implicit state.

Additionally, we could deduce little information about the implementation code because error conditions are a major source of semantic information, but all error checks in the code happened very early in the code paths.

Error-checking early is a well-recognized best secure coding practice: any unexpected input should be rejected as soon as possible in order to prevent it from affecting the system state, possibly introducing a vulnerability. Therefore, we expect that most systems where security is a concern will do likewise.

We also argue that implicit state is not atypical. Caches represent implicit state, as do data records with id fields. Implementations that use such are not uncommon in medium or large systems. Although it is premature to conclude a-priori that feedback-directed testing is not feasible for these systems, we feel that our results do raise doubts.

6 Conclusion

We have described results of two case studies designed to satisfy testing requirements as mandated by higher levels of Common Criteria for security evaluations. These requirements include 1) Demonstrable independence of individual test cases 2) Demonstrable coverage of both user level specifications and implementation code.

We conducted two case studies, both based on the same highly secure smart card operating system. In the first case study, we developed an approach to automatically generate self-contained test cases based on specifications. We used such self-contained test cases to demonstrate that self-contained test cases are independent and do not lead to fault masking.

In the second case study, our goal was to demonstrate adequate code coverage, while retaining the ability to generate expected outputs based on specifications. We first established mappings between the specification and implementation code elements. We then used test cases generated from the user level specification to identify the executed code elements and attempted to use static analysis to map the unexecuted code elements to the corresponding elements in the user level specification. In this second case study, we found that, given a sufficiently expressive user level specification, and a test generation system that is able to effectively use such a specification, the resulting tests will cover the vast majority of the code branches that are able to be covered. Therefore, the benefit of a feedback-directed system will be limited. We further provide evidence that the static analysis required to generate feedback in these cases tends to be difficult, involving inferring the semantics of the internal implementation of data structures. In particular, we observed that the internal states at the implementation level in a high security application pose significant challenges to this mapping process.

References

[1] "Common criteria for information technology security evaluation, parts 1, 2, and 3," Tech. Rep. Version 3.1, Revision 1, CCMB–2006–09–001, CCMB–2006–09–002, and CCMB–2006–09–003, Sep. 2006, http://www.commoncriteriaportal.org/thecc.html.

[2] M. Kaplan, T. Klinger, A. Paradkar, A. Sinha, C. Williams, and C. Yilmaz, "Less is more: A minimalistic approach to uml model-based conformance test generation," in *ICST '08*, 2008, pp. 82–91.

[3] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[4] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *ESEC/FSE*, 2005, pp. 263–272.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," in *CCS*, 2006.

[6] A. Andrews, R. B. France, S. Ghosh, and G. Craig, "Test Adequacy Criteria for UML Design Models," *JSTVR*, vol. 13, no. 2, pp. 95–127, 2003.

[7] L. Briand, Y. Labiche, and Y. Wang, "An investigation of graph-based class integration test order strategies," *Transactions on Software Engineering*, vol. 29, no. 6, pp. 1–37, 2003.

[8] D. Kung *et al.*, ""On Regression Testing of Object-Oriented Programs"," *Journal of Systems and Software, Vol. 32*, pp. 21–40, Jan. 1996.

[9] Y. Labiche, P. Thvenod-Fosse, H. Waeselynck, and M.-H. Durand, "Testing levels for object-oriented software," in *Proceedings of International Conference on Software Engineering*, 2001.

[10] Y. L. Traon, "Efficient object-oriented integration and regression testing," *IEEE Transactions on Reliability*, vol. 49, no. 1, pp. 12–25, 2000.

[11] K. C. Tai and F. Daniels, "Interclass test order for object-oriented software," *Journal of Object Oriented Programming*, vol. 12, no. 4, pp. 18–25, 1999.

[12] R. Majumdar and R.-G. Xu, "Directed test generation using symbolic grammars," in *ASE*, 2007, pp. 134–143.

[13] P. Godefroid, A. Kieżun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI*, 2008.

[14] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE*, 2007, pp. 416–426.

[15] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *ISSTA*, 2007, pp. 151–162.

[16] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *JSTVR*, vol. 9, pp. 263–282, 1999.

[17] P. Mcminn, "Search-based software test data generation: A survey," *JSTVR*, vol. 14, p. 2004, 2004.

[18] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler, "Automatically generating malicious disks using symbolic execution," in *SP'2008*, 2006, pp. 243–257.

[19] G. Friedman, A. Hartman, K. Nagin, and T. Shiran, "Projected state machine coverage for softare testing," in *ISSTA*, 2002, pp. 134–143.

[20] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes, ""Extracting a finite state machine from ASM specification"," in *ISSTA*, 2002, pp. 112–122.

[21] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *J. STVR*, vol. 13, pp. 25–53, 2003.

[22] A. Petrenko, S. Boroday, and R. Groz, ""Confirming Configurations in EFSM Testing"," *IEEE TSE*, vol. 30, no. 1, pp. 29–42, Jan. 2004.

[23] D. C. Toll, P. A. Karger, E. R. Palmer, S. K. McIntosh, and S. Weber, "The Caernarvon secure embedded operating system," *Operating Systems Review*, vol. 42, no. 1, pp. 32–39, 2008.

[24] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM TOPLAS*, vol. 24, no. 3, pp. 217–298, 2002.

[25] L. Prechelt, "Why we need an explicit forum for negative results," *Journal of Universal Computer Science*, vol. 3, no. 9, 1997.