RC23717 (W0509-057) September 16, 2005
Computer Science

IBM Research Report

Chicken & Egg: Dependencies in Security Testing and Compliance with Common Criteria Evaluations

Amit Paradkar, Suzanne McIntosh, Sam Weber,
David Toll, Paul Karger, Matt Kaplan
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Chicken & Egg: Dependencies in Security Testing and Compliance with Common Criteria Evaluations

Amit Paradkar, Suzanne McIntosh, Sam Weber, David Toll, Paul Karger, Matt Kaplan
IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532, USA
e-mail : {paradkar, skranjac, toll, mmk}@us.ibm.com, {samweber, karger}@watson.ibm.com

Abstract

Common Criterion security evaluations require an analysis of test dependencies present (if any) during the testing activities of an application under evaluation. Such analysis is required to ensure that no fault masking occurs. In this paper, we present 1) a formalization of the notion of test dependencies from fault masking perspective in terms of a test dependency graph (TDG), 2) a model based approach for derivation of a TDG for a set of use cases, 3) an algorithm which derives a test order from a TDG to minimize the debugging cost, and 4) results from a case study using a secure smart card operating system. Our results indicate that fault masking in the presence of test dependencies is not a serious concern.

1. Introduction

Explosive growth in worms and viruses has made IT security a critical issue both for vendors and consumers of IT. Many consumers, including governments of several nations, require IT vendors to demonstrate security compliance of applications that they provide. An international standard, called the Common Criteria (**CC**), which supports different levels of assurance has been developed to evaluate a given IT application. The Common Criteria Scheme [1] enables consumers to obtain an impartial security evaluation of an IT product by an independent entity. The specific IT product being evaluated is referred to as the Target of Evaluation (TOE). The security requirements for that product are described in its security target.

CC security evaluation includes an analysis of the IT product and the testing of the product for conformance to a set of security evaluation requirements. These evaluation criteria require an development organization to furnish evidence of security compliance along seven dimensions (called assurance classes) as follows Configuration Management (ACM), Delivery and Operation (ADO), Development (ADV), Guidance Documents (AGD), Lifecycle Support (ALC), Testing (ATE), and Vulnerability Analysis (AVA).

Each assurance class is further divided into families and components which detail the compliance requirements. The **ATE** assurance class, for example, is divided into 4 families as follows:

1. Coverage (ATE_COV): Requires evidence that testing activities have covered each TOE security function adequately
2. Depth (ATE_DEP): Requires evidence that the testing activities have covered TOE security functions to appropriate depth (such as high level or low level design)
3. Functional Testing (ATE_FUN): Requires evidence that the testing activities have covered functionality of each TOE security function adequately.
4. Independent Testing (ATE_IND): Imposes requirements on the independent evaluators

Each family describes contents and presentation of the required evidence. One aspect of the ATE_FUN family is concerned with test dependency as quoted in the CC document [1] "... Ordering dependencies are relevant when the successful execution of a particular test depends upon the existence of a particular state. For example, this might require that test A be executed immediately before test B, since the state resulting from the successful execution of test A is a prerequisite for the successful execution of test B. Thus, failure of test B could be related to a problem with the ordering dependencies. In the above example, test B could fail because test C (rather than test A) was executed immediately before it, or the failure of test B could be related to a failure of test A."

The specific requirements concerning test ordering are (as quoted from [1][1]): “**ATE_FUN.2** Ordered functional testing Objectives. The objective is for the developer to demonstrate that all security functions perform as specified. The developer is required to perform testing and to provide test documentation. In this component, an additional objective is to ensure that testing is structured such as to avoid circular arguments about the correctness of the portions of the TSF being tested.

ATE_FUN.2.6C The test documentation shall include an analysis of the test procedure ordering dependencies.”

These 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. For applications which provide *services* to other applications, the problem of test dependencies can be cast in terms of ordering dependencies among services. The ordering is imposed because of the system state being manipulated by the provided services. Provided services may depend on a particular system state to operate correctly and depend on other services to achieve the desired system state. Such dependencies imply that *integration testing* of the provided services may need to be carried out in certain order to ensure that the test case are *self-contained*, to minimize the probability of fault masking, and to reduce the cost of debugging process.

The problem of dependency and order in the context of integration testing of object-oriented (OO) software has been studied extensively [4, 5, 9, 11, 10]. However, in all the previous works the emphasis is on minimizing the cost of stub development, and not on reducing fault masking. The dependencies are defined in terms of inheritance, aggregation, and association relationships among the classes being tested. However, this notion of dependency is not suitable for a set of services. In this paper, we provide a more appropriate formulation of test dependency among a group of services. We use this formulation for a model based approach to perform the analysis required in **ATE_FUN.2.6C** above. We use Unified Modeling Language (UML) to describe the external behavior of the services (and treat a service like an UML use case). We define 3 kinds of dependencies between a pair of services:

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

[1] The changes proposed for the Common Criteria V3 [2] do not materially affect these requirements.

Additionally, the specific contributions of this paper are:

- Concept of a *Test Dependency Graph* (TDG) which encapsulates the dependency relationships among a set of services.
- A technique to derive a TDG based on a UML behavior model of the services.
- A technique to derive a test order for services based on a two-phase topological sort of the TDG.
- Results of a case study using a smart card operating system component demonstrating that fault masking is reduced with the *self-contained* test cases created using the test order.

The rest of this paper is organized as follows. Section 2 describes the concepts involved in the UML behavior model for the services along with an example based on file system component of a smart card operating system. Section 3 defines 1) various dependency relationships between a pair of services, and 2) the test dependency graph itself. Section 3 also describes algorithm to derive a test order based on TDG. Section 4 describes the results of a case study using the file system component. Section 5 reviews the prior results in integration test order which are related to ours. Finally, Section 6 summarizes our conclusions and indicates areas of future work.

2. Preliminaries

2.1. Terminology

We use Unified Modeling Language (UML) [3] to describe the behavior of the services provided by an application. The behavior model used in this paper consists of 2 artifacts: 1) Domain Model, and 2) Use case Model.

A *Domain model* $\mathcal{D}$ consists of conceptual entities and relationships thereamong that are manipulated by the provided services. These concepts are represented using the Class diagram artifact of UML. A UML Class diagram consists of classes in the system each describing a conceptual domain entity. A class has a list of attributes and associated types. Two classes may be related to each other in a variety of ways. *Inheritance* relationship captures an *is-a* relationship between two classes and is depicted by a hollow arrowhead in the diagram notation. An *association* relation exists if the two entities are logically related to each other, and is depicted by a plain line in the diagram.

A system state consists of concrete instance of a domain model. A concrete instance of a *Class* is called an *Object*, and that of an *Association* is called a *Link*.

A *Use case Model* in UML model describes the behavior of an individual use cases. Let $\mathcal{U}$ be the set of use cases that an application provides. Each $U \in \mathcal{U}$ takes parameters $\mathcal{P}_U = \mathcal{I}_U \cup \mathcal{O}_U$, where $\mathcal{I}_U$ is the set of input parameters and $\mathcal{O}_U$ is the set of output parameters, and $\mathcal{I}_U \cap \mathcal{O}_U = \emptyset$; and produces exactly one of the results $\mathcal{R}_U = \mathcal{X}_U \cup \mathcal{N}_U$ for any invocation. $\mathcal{X}_U$ is the set of *abnormal* results which raise *exceptions* to the environment and $\mathcal{N}_U$ is the set of *normal* results. Each $R_U \in \mathcal{R}_U$ has a *guard* condition, $G_{R_U}(\mathcal{I}_U, \mathcal{D})$, which is a predicate on the input parameters and entities from the domain model.

Each $R_U \in \mathcal{R}_U$ has an associated postcondition $\mathcal{A}_{R_U}(\mathcal{P}_U, \mathcal{D})$ expressed as a list of *imperative* actions. Some actions affect entities in the domain model, while others assign values to the output parameters. Actions which affect the domain model entities follow a Create, Read, Update, and Delete (CRUD) paradigm for both instances and links in the domain model. A *Create* action creates an instance of an object or a link (along with appropriate initialization of the corresponding attributes). A *Read* action assigns values of a domain model entity to an output parameter. An *Update* action modifies an instance (by changing its attributes). A *Delete* action deletes an object or a link.

2.2. An Example Application

We will illustrate the above terminology and our techniques with a file system component of the Caernarvon operating system software [7]. Caernarvon is a high assurance operating system currently under development at IBM T.J. Watson Research Center for use in smart card applications. Smart card chips are equipped with a microprocessor, transient and persistent memory, and certain other peripheral devices - in particular an input/output and communication device. Smart card chips are commonly used in banking applications such as electronic purse/debit cards. They are also used in securing information such as may be found on national identification cards and, more recently, passports.

The Caernarvon operating system is different from most smart card operating systems, because it requires a processor chip that supports both user and supervisor modes and that supports memory protection. In traditional smart card operating systems, all code running on the smart card must be trusted, because it can touch all of memory. However, Caernarvon supports downloading of applications that may not be fully trusted or that may be mutually suspicious and require protection from each other.

Traditional smart cards provide services based on application protocol data units (APDUs) that are used to communicate between the smart card and the reader. However, in the Caernarvon operating system, the services provided are supervisor calls that are used to communicate between applications programs and the trusted operating system kernel. The APDUs are implemented by applications program which may or may not be trusted to be secure.

Thus, at the very highest 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 Crypto System services, the Key System services, the File System services, 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.

Figure 1 shows the domain model for the file system component of Caernarvon. 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 `Directorys` and `Files`.). This is depicted by the parent-children association between the two classes. A `FileSystemObject` consists of blocks of `Data` (represented as a *Sequence*). Also, A `FileSystemObject` may be referred to by a `Handle` object.

Figure 2 describes some of the file system services modeled as use cases. For brevity, we describe only the essential behavior, leaving out many details. For example, use case `Create Directory` takes two input arguments: `parentName` and `pName`. It then checks if a `Directory` instance with the name equal to `parentName` exists in the current system state. If such an instance exists, it creates a new `Directory` instance with name equal to the parameter `pName`, and creates a `parent` and `child` links between the parent `Directory` and the newly created `Directory` instances. In what follows, we refer to the file system behavior model (consisting of the Domain model and the use case model) as `FileSystemSpec`.

3. Test Dependency Graph

3.1. Terminology

A notion of Test Dependency Graph, or TDG, is defined below to capture the relationships among the use cases of a system.

Definition 1 *A use case U is a* **constructor** *of a Class C (or Association A) if there exists a create action for an instance of Class C (Association A resp.) in $\mathcal{A}_{R_U}(\mathcal{P}_U, \mathcal{D})$ for some result R_U of U.*

For example, in `FileSystemSpec` use case `Create Directory` is a *Creator* of `Directory` class and

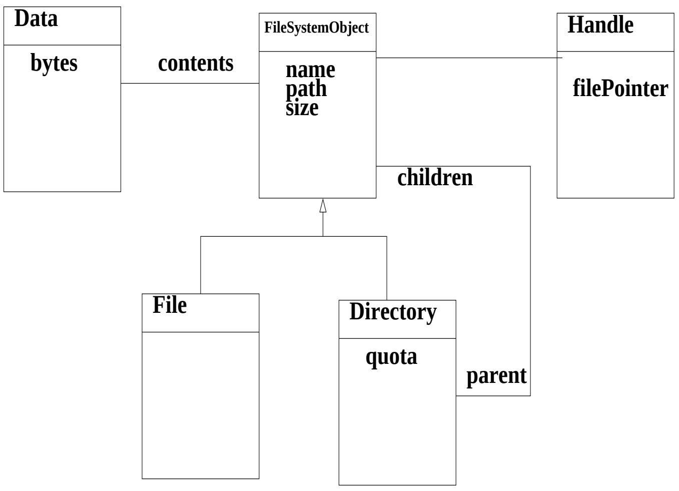

Figure 1. Domain Model for `FileSystemSpec`

`Directory - FileSystemObject` association, while use case `File Open` is a *Creator* of `Handle` class, and the `FileSystemObject-Handle` association.

Definition 2 *A use case U is a* **destructor** *of a Class C (or Association A) if there exists a delete action for an instance of Class C (Association A resp.) in $\mathcal{A}_{R_U}(\mathcal{P}_U, \mathcal{D})$ for some result R_U of U.*

For example, use case `Delete File` is a *destructor* of class `File` and association `FileSystemObject -Directory` in `FileSystemSpec`.

Definition 3 *A use case U is a* **reader** *of a Class C (or Association A) if there exists a read action for an instance of Class C (Association A resp.) in $\mathcal{A}_{R_U}(\mathcal{P}_U, \mathcal{D})$ for some result R_U of U.*

For example, use case `Open File` is a *reader* of class `Handle`, and use case `File Read` is a *reader* of Class `Data`.

Definition 4 *A use case U is a* **user** *of a Class C (or Association A) if there exists a reference to an instance of Class C (Association A resp.) in $G_{R_U}(\mathcal{I}_U, \mathcal{D})$ for some result R_U of U.*

For example, use case `File Create` is an *user* of `Directory` class since, its guard condition checks to see if an appropriate instance of `Directory` exists.

Definition 5 *The relation SetsUp : $\mathcal{U} \times \mathcal{U} \rightarrow \{true, false\}$ for a pair of use cases $U_1, U_2 \in \mathcal{U}$, denoted SetsUp(U_1, U_2), is true iff there exists an entity (either a class or an association) $E \in \mathcal{D}$ such that U_1 is a creator of E and U_2 is an user of E.*

The relation `SetsUp` captures the set up dependency, where one use case prepares the system state by creating entities (objects or links) that would be needed by others. For example, `SetsUp (Create Directory, Create File)` is *true* since `Create Directory` is a creator of `Directory` which is used by `Create File`.

We define analogous relations *Verifies* and *CleansUp* as follows.

Definition 6 *The relation Verifies : $\mathcal{U} \times \mathcal{U} \rightarrow \{true, false\}$ for a pair of use cases $U_1, U_2 \in \mathcal{U}$, denoted Verifies(U_1, U_2), is true iff there exists an entity (either a class or an association) $E \in \mathcal{D}$ such that U_1 is either a reader or a user of E and U_2 is a creator of E.*

The relation `Verifies` captures the verification dependency, where one use case demonstrates the updates to the system state in terms of entity creation by using another use case to externalize the evidence of its existence. For example, `Verifies (Read File, Write File)` is *true* since `Read File` demonstrates the impact of `Write File` to the environment through a read action. Similarly, `Verifies (Open File, Create File)` is true since, a file created by `Create File` can be used by an `Open File` to demonstrate its successful creation.

Definition 7 *The relation CleansUp : $\mathcal{U} \times \mathcal{U} \rightarrow \{true, false\}$ for a pair of use cases $U_1, U_2 \in \mathcal{U}$, denoted CleansUp(U_1, U_2), is true iff there exists an entity (either a class or an association) $E \in \mathcal{D}$ such that U_1 is a destructor of E and U_2 is a creator of E.*

The relation `CleansUp` captures the clean up dependency, where one use case deletes the entities created by another use case. This is useful in the testing prac-

```
<COMPONENT> File System
  <use case> Create Directory
    in parentName
    in Name
    RESULT rOK IF
      a directory with name=parentName exists
    post {Create a new Directory Instance}
    {Create a children link}
  </use case>

  <use case> Remove Directory
    in pName
    RESULT rOK IF
      a directory with name=pName exists
    post {Remove the directory Instance}
  </use case>

  <use case> Create File
    in parentName
    in Name
    RESULT rOK IF
      a directory with name=parentName exists
    post {Create a new File Instance}
    {Create a children link}
  </use case>

  <use case> Delete File
    in pName
    RESULT rOK IF
      a file with name=pName exists
    post {Delete the found File Instance}
    </use case>

<use case> Open File
  in pName
  out handle
  RESULT rOK IF
    a fileObject with name=pName exists
  post {Create a new Handle Instance}
  {Create a handles link}
  {Return the handle instance}
</use case>

<use case> Close File
  in handle
  RESULT rOK IF
    pHandle exists in the Set of Handles
  post {Delete the pHandle Instance}
</use case>

<use case> Write File
  in pHandle
  in buffer
  RESULT rOK IF
    a handle = pHandle exists
  post {Create a new Contents Instance}
  {Set the blocks attribute to buffer}
  {Create a contents link}
</use case>

<use case> Read File
  in pHandle
  out buffer
  RESULT rOK IF
    a handle = pHandle exists
  post
  {Copy the blocks into the buffer output}
</use case>
```

Figure 2. Use case Model for `FileSystemSpec`

tice to have a self contained set of test cases, which restore any modifications to the persistent system state back to the original values. For example, `CleansUp (Create Directory, Remove Directory)` is *true* since `Create Directory` is a creator of `Directory` which is deleted by `Remove Directory`.

Definition 8 *An edge labeled diagraph $G = (V, E, L)$ is a directed graph, where $V = \{V_1, \ldots, V_n\}$ is a finite set of nodes, $L = \{L_1, \ldots, L_k\}$ is a finite set of labels, and $E \subseteq V \times V \times L$ is a finite set of edges.*

Definition 9 *The TDG for a set of use cases is an edge labeled diagraph $G = (V, E, L)$, where V is the set of use cases, $L = \{S, V, C\}$, and $E = E_S \cup E_V \cup E_C$ is the set of edges as defined below.*

Definition 10 *$E_S \subseteq V \times V \times L$ is the set of directed edges representing the set up relationship among the use cases. For any two use cases $U_1, U_2 \in \mathcal{U}$, $< U_1, U_2, S >\in E_S$ if SetsUp(U_1, U_2) is* true.

Definition 11 *$E_V \subseteq V \times V \times L$ is the set of directed edges representing the verification relationship among the use cases. For any two use cases $U_1, U_2 \in \mathcal{U}$, $< U_1, U_2, V >\in E_V$ if Verifies(U_1, U_2) is* true.

Definition 12 *$E_C \subseteq V \times V \times L$ is the set of directed edges representing the clean up relationship among the use cases. For any two use cases $U_1, U_2 \in \mathcal{U}$, $< U_1, U_2, C >\in E_C$ if CleansUp(U_1, U_2)* true.

Figure 3 shows the result of applying the TDG definitions to the File System component behavior model.

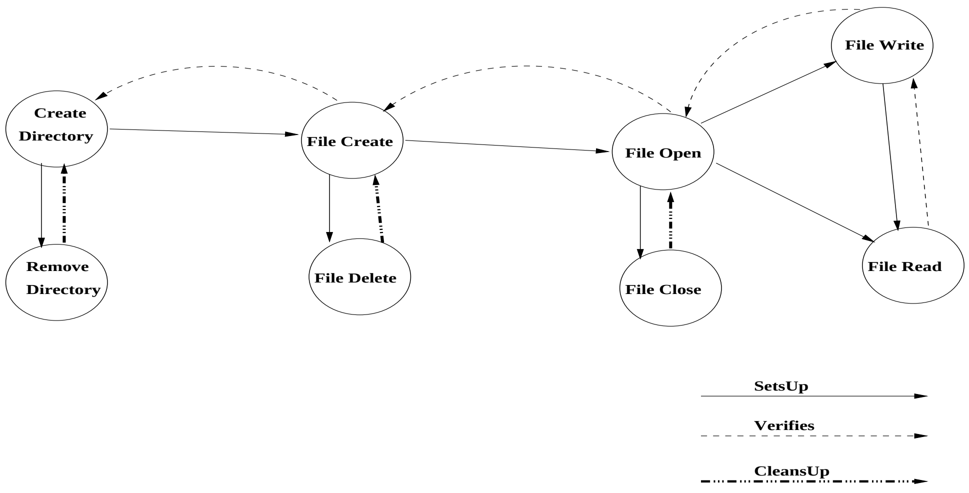

Figure 3. Test Dependency Graph for `FileSystemSpec`

3.2. Deriving Test Orders from TDG

The algorithm for finding test dependency order operates in two phases. In the first phase, the original TDG is converted into an acyclic directed graph by identifying its Strongly Connected Components (SCCs) [6]. A topological sort of the resulting acyclic graph yields a major order of traversal. For the TDG in Figure 3, the entire graph is a single SCC.

In the second phase, each SCC identified in the first phase is converted into an acyclic graph by removing certain edges from the SCC. A second topological sort of the resulting acyclic graphs yields minor test order.

To break the cycles, we delete an edge which is not of `SetsUp` kind. With this algorithm, the derived test order is as follows:

`Create Directory`, (`Create File` || `Remove Directory`), (`Open File` || `Delete File`), `Write File`, `Read File`, where || implies that any choice would be correct.

Along with the test order, our algorithm also produces a list of the verification and clean up services that could be used to create a *self-contained* test case for a given service.

4. Experimental Evaluation

The experiments described in this section were designed to enable us to search for evidence of fault masking in the presence of multiple faults using a test suite of *self-contained* test cases. These test cases were created manually prior to the development of the test order formulation described in Section 3.1, but approximate the test order derived for the file system services in Section 3.2. In support of our experiments, we injected seeded faults into the stable implementation of Caernarvon available to us.

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 services, *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.

In the next section, we provide an overview of the Caernarvon implementation along with the associated test environment in which we performed our experiments. We then describe the nature of the injected faults and the criteria used in vetting single and combined faults. Finally, we describe the fault injection experiments performed and results obtained.

4.1. Target Environment Overview

The implementation of the File System component of the Caernarvon operating system consists of approximately 16K lines of `C` code (including comments and embedded documentation) of which approximately 5K are executable `C` statements. The Caernarvon test environment consists of a test bench and test scripts written in the scripting language `Ruby`. The test scripts are the implementations of numerous *self-contained* test cases intended to cover both the successful and failing behavior of various services. Each test case is composed of four basic stages:

1. The setup stage establishes the environment prerequisite for testing of the main objective.

2. The objective testing stage carries out the test objective.
3. The verification stage compares results obtained to expected results.
4. The cleanup stage returns the system to the state which existed prior to the setup stage.

The test system runs these scripted test scenarios and then captures the results for post-test analysis.

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

1. Insertion 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.

CFIT 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. CFIT collects test results, unambiguously associating the results with the faults injected.

4.2. Fault Injection Experiments

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 boot-up sequence. 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 our 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 or update the file pointer properly
- trap a read beyond the end of file

In Table 1, we illustrate some of the injected faults. 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 1 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 1 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 102 fault pairs that did not meet our criteria. This left us with 207 paired fault BIFs (135 inter-service, 72 intra-service) for further analysis.

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 1. Faults Injected into File System Implementation

4.3. Fault Masking Analysis Results

Endeavoring to find evidence 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.

5. Related Work

One important problem when integrating and testing object-oriented software is 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 [4, 8, 9, 11, 10]. The objective of all 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 [10] 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. [11] 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 [10], point out that association relationships are usually the weakest links in a class diagram, i.e., the links involving the fewest dependencies and therefore the least stub complexity if broken.

6. Conclusions and Future Work

Common Criterion security evaluations require an analysis of test dependencies present (if any) during the testing activities of an application under evaluation. Such analysis is required to ensure that the test cases are *self-contained*, and no fault masking occurs. In this paper, we have presented 1) a formalization of the notion of test dependencies from fault masking perspective in terms of a test dependency graph (TDG), 2) a model based approach for derivation of a TDG for a set of use cases, 3) an algorithm which derives a test order from a TDG to minimize the debugging cost, and 4) results from a case study using a secure smart card operating system. Our results indicate that fault masking in the presence of test dependencies is not a serious concern.

One of the CC requirements at higher levels of evaluation is to demonstrate that all the implementation code is exercised thoroughly. However, the coverage adequacy criterion is not clearly specified. We would like to make this criterion more precise. To that end, we plan to conduct experiments to measure coverage of the code entities (*e.g.* statements, branches) in Caernarvon implementation using the existing test suite, and explore the relationship between code coverage and detected faults. We would also like to investigate the impact of set up, verification, and clean up aspects of a test case on the relationship between measured code coverage and fault detection effectiveness.

References

[1] Information technology - security techniques – evaluation criteria for IT security – part 3: Security assurance requirements, 1999. ISO/IEC 15408-3, International Standards Organization.

[2] Common criteria for information technology security evaluation - part 3: Security assurance components, 2005. Version 3.0, Revision 2, CCMB-2005-07-003.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[4] L. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategies. *Transactions on Software Engineering*, 29(6):1–37, 2003.

[5] L. Briand, Y. Labiche, and Y. Wang. Towards a comprehensive and systematic methodology for class integration testing. In *Proc. of Int. Symp. on Software Reliability Engineering*, Nov. 2003.

[6] N. Deo. *Graph Theory and Applications to Engineering and Computer Science*. 1974.

[7] P. A. Karger, V. R. Austel, and D. Toll. Using mandatory secrecy and integrity for business to business applications on mobile devices. In *Workshop on Innovations in Strong Access Control*, pages 25–27, 2000. http://www.acsac.org/sactac/wisac00/wed0830.karger.pdf.

[8] D. Kung et al. "On Regression Testing of Object-Oriented Programs". *Journal of Systems and Software, Vol. 32*, pages 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] K. C. Tai and F. Daniels. Interclass test order for object-oriented software. *Journal of Object Oriented Programming*, 12(4):18–25, 1999.

[11] Y. L. Traon. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability*, 49(1):12–25, 2000.