Consistency Analysis of Authorization Hook Placement in the Linux Security Modules Framework

TRENT JAEGER
IBM T. J. Watson Research Center
ANTONY EDWARDS
Symbian Ltd.
and
XIAOLAN ZHANG
IBM T. J. Watson Research Center

We present a consistency analysis approach to assist the Linux community in verifying the correctness of authorization hook placement in the Linux Security Modules (LSM) framework. The LSM framework consists of a set of authorization hooks inserted into the Linux kernel to enable additional authorizations to be performed (e.g., for mandatory access control). When compared to system call interposition, authorization within the kernel has both security and performance advantages, but it is more difficult to verify that placement of the LSM hooks ensures that all the kernel's security-sensitive operations are authorized. Static analysis has been used previously to verified mediation (i.e., that some hook mediates access to a security-sensitive operation), but that work did not determine whether the necessary set of authorizations were checked. In this paper, we develop an approach to test the *consistency* of the relationships between security-sensitive operations and LSM hooks. The idea is that whenever a security-sensitive operation is performed as part of specifiable event, a particular set of LSM hooks must have mediated that operation. This work demonstrates that the number of events that impact consistency is manageable and that the notion of consistency is useful for verifying correctness. We describe our consistency approach for performing verification, the implementation of run-time tools that implement this approach, the anomalous situations found in an LSM-patched Linux 2.4.16 kernel, and an implementation of a static analysis version of this approach.

Categories and Subject Descriptors: D.2.9 [**Software Engineering**]: Management—*software configuration management*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*unauthorized access*

An earlier version containing some portions of this paper appeared as "Runtime Verification of Authorization Hook Placement in the Linux Security Modules Framework" in the *Proceeding of the 9th ACM Conference on Computer and Communications Security*, pages 225–234, November 2002.

Authors' addresses: Trent Jaeger and Xiaolan Zhang, 19 Skyline Drive, Hawthorne, NY, USA 10532, email: {jaegert, cxzhang}@us.ibm.com; Antony Edwards, 2-6 Boundary Row, London, SE1 8HP, United Kingdom, email: antonye@cse.unsw.edu.au. This work was done while the author was on an internship at the IBM T.J. Watson Research Center.

General Terms: Design, Management, Security

Additional Key Words and Phrases: access control models, authorization mechanisms, role-based access control

1. INTRODUCTION

The Linux Security Modules (LSM) project aims to provide a generic framework from which a wide variety of authorization mechanisms and policies can be enforced. Such a framework would enable developers to implement authorization modules of their choosing for the Linux kernel. System administrators can then select the module that best enforces their system's security policy. Typically, the aim is to enforce a mandatory access control policy in addition to the traditional UNIX discretionary policy to enable containment of compromised system services. The LSM framework has been accepted into the mainline Linux kernel (`www.kernel.org`) as of version 2.6 along with the SELinux and Linux capability LSMs.

The LSM framework implements a reference monitor interface [Anderson 1972] by inserting a set of authorization hooks as necessary the Linux kernel. These hooks define the types of authorizations that a module can enforce and their locations. Placing the hooks in the kernel itself rather than at the system call boundary has security and performance advantages. The main problem is that in several system calls the name of an object, rather than its reference, is passed from the user-level process to the kernel (e.g., open). First, system call interposition must resolve the object name to an object reference to authorize it. Since the kernel also performs this resolution, so there is an unnecessary performance overhead. Second, and more importantly, the mapping between the object name and the object reference may be changed between the authorization and the kernel resolution, resulting in an unauthorized access. Thus, system call interposition is said to be susceptible to time-of-check-to-time-of-use (TOCTTOU) attacks [Bishop and Dilger 1996], where another object is swapped for the authorized object after authorization.

One of the key aspects of a reference monitor interface is that it ensures that all *controlled operations* (i.e., those operations whose control is necessary for security) are authorized before they are run. While placing the LSM reference monitor's authorization hooks in the kernel can improve security, it is more difficult to determine whether the hooks mediate and authorize all controlled operations. The system call interface is a nice mediation point because all the kernel's controlled operations (i.e., operations that access security-sensitive data) *must* eventually go through this interface. Inside the kernel, there is no obvious analogue for the system call interface. Any kernel function can contain accesses to one or more security-sensitive data structures. Thus, any mediation interface is at a lower level of abstraction (e.g., inode member access). In addition to mediation, it is also necessary to ensure that the proper access control policy (e.g., write data) is enforced for each security-sensitive operation. If there is a mismatch between the policy enforced and the controlled operations that are executed under that policy, unauthorized operations can be executed.

We believe that manual verification of the correct authorization of a low-level mediation interface is impractical.

Recently, a variety of work has demonstrated the possible effectiveness of static source-code analysis for the discovery of security bugs [Engler et al. 2000; Larochelle and Evans 2001; Shankar et al. 2001] and even the verification of some security properties within some reasonable assumptions [Chen and Wagner 2002; Zhang et al. 2002].

In other work, we have developed a static analysis approach that enables comprehensive verification that the variables used in security-sensitive operations have been authorized [Zhang et al. 2002]. However, it is more difficult to determine if all variables used in security-sensitive operations have been authorized *for all the necessary operations*. First, we need a model to help us predict when we have identified the necessary authorizations. Second, we need an analysis approach to enable us to verify this model. One insight that we found useful is that many LSM authorization hooks are correctly placed, so inconsistencies in authorization requirements are often indicative of a problem. Another insight we have found is that consistency in authorization is context-dependent, so we need some way to express and test contexts in which we expect consistent authorization. We found it easier to explore possible analysis options using a run-time analysis tool, so we describe the nature of such a tool here. However, we have ultimately found that this approach can also be leveraged by a static analysis approach, so we briefly describe this prototype. Thus, we provide insight into developing analyses via run-time tools and examining their application in static analysis tools.

In this paper, we present a consistency analysis approach to assist the LSM community and Linux kernel developers in verifying that the LSM authorization hooks completely authorize accesses. We also present implementation of this approach, using both run-time and static analysis techniques. In both cases, the implementation consists of two parts: (1) a data collection tool that generates system logs containing the events relevant to measuring consistency and (2) a consistency analysis tool that identifies the consistency between the controlled operations and LSM hooks. System log generation is done either via run-time instrumentation of the Linux kernel or by static analysis of the Linux kernel source code. Run-time collection is accurate, but misses many of the possible execution paths, so we have implemented a static analysis collection mechanism that generates compatible logs. The consistency analysis finds hook placement errors from this collected data by identifying inconsistencies where consistent authorizations are expected. We have designed a filtering language for describing contexts in which consistent authorization is expected. Our analysis tools generate two different representations that we used to find inconsistencies: (1) *authorization graphs* that display the consistency between the execution of a controlled operations and its authorizations, and (2) *sensitivity class lists* that show the attributes of controlled operations to which the authorization consistency is sensitive.

Using this approach, we have found three bugs in LSM hook placement in the file system that have since been fixed, and another anomaly that resulted in significant discussion. While the approach we use at present is not complete

(i.e., some bugs may be missed), we are encouraged by our ability to find bugs using these tools. We demonstrate the use of these tools on a LSM-patched Linux kernel version 2.4.16.

The remainder of the paper is structured as follows. In Section 2, we define the general hook placement problem. In Section 3, we develop an approach to solving the general hook placement problem. In Section 4, we outline the implementation of the run-time data collection and consistency analysis tools and discuss the analyses performed and their results. In Section 5, we describe how static analysis can be used for log collection. In Section 6, we describe issues related to the use of such tools, such as regression testing. In Section 7, we conclude and describe future work.

2. GENERAL HOOK PLACEMENT PROBLEMS

2.1 Concepts

We identify the following key concepts in the construction of an authorization framework:

—**Security-sensitive Operations:** These are the operations that impact the security of the system.

—**Controlled Operations:** A subset of security-sensitive operations that mediate access to all other security-sensitive operations. These operations define a *mediation interface*.

—**Authorization Hooks:** These are the authorization checks in the system (e.g., the LSM-patched Linux kernel).

—**Policy Operations:** These are the conceptual operations authorized by the authorization hooks.

Correct authorization hook placement must ensure that the *authorization hooks* authorize all *security-sensitive operations*. Such authorizations test whether the system's authorization policy permits the requesting principal to execute the particular security-sensitive operations. It is more convenient to express authorization policy at a higher level (e.g., file read or write), so rather than authorizing the individual security-sensitive operations we authorize conceptual operations, which we call *policy operations*. Further, since the number of security-sensitive operations can be large, it is preferable to authorize them once at an interface that mediates all the security-sensitive operations. The set of controlled operations defines such a mediation interface. Thus, we define our problem to verify that all controlled operations are authorized for the expected policy operations using the LSM authorization hooks.

Identifying the controlled operations is more difficult for the in-kernel mediation of LSM than for the system call mediation mechanisms of the past. As shown in Figure 1, the system call interface is well known for providing mediation of all the security-sensitive operations in the system call. Therefore, the system call interface can be used both as the controlled operations and the policy operations.

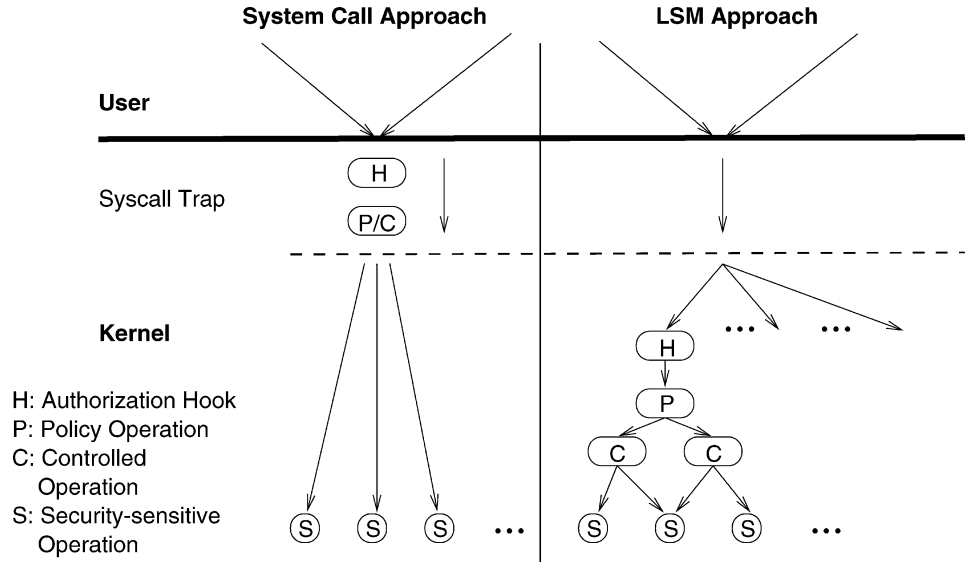

Fig. 1. Comparison of concepts between system call interposition framework and LSM.

When authorization hooks are inserted in the kernel, a mediation interface is no longer obvious, so the controlled operations and their mapping to policy operations are no longer so easy to identify. For example, rather than verifying file open for write access at the system call interface, the LSM authorizations for directory (exec), link (follow link), and ultimately, the file (write) are performed at the time these operations are to be done. This approach has the benefits of eliminating susceptibility to TOCTTOU attacks [Bishop and Dilger 1996] and redundant authorization processing, but in order to verify the hook placement more work is necessary to identify the controlled operations, the policy operations they correspond to, and verify that the authorization hooks authorize them properly.

2.2 Relationships to Verify

Figure 2 shows the relationships between the concepts.

(1) **Identify Controlled Operations:** Find the set of operations that define a mediation interface through which all security-sensitive operations are accessed.
(2) **Determine Authorization Requirements:** For each controlled operation, identify the authorization requirements (i.e., policy) that must be authorized by the LSM hooks.
(3) **Verify Complete Authorization:** For each controlled operation, verify that the correct authorization requirements are authorized by LSM hooks.
(4) **Verify Hook Placement Clarity:** Controlled operations implementing a policy operation should be easily identifiable from their authorization

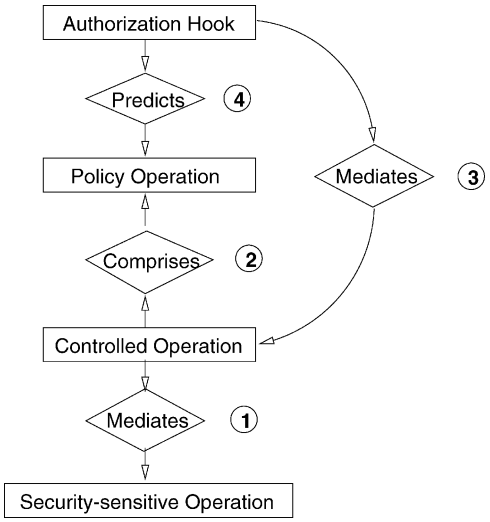

Fig. 2. Relationships between the authorization concepts. The verification problems are to: (1) identify controlled operations; (2) determine authorization requirements; (3) verify complete authorization; and (4) verify hook placement clarity.

hooks. Otherwise, even trivial changes to the source may render a hook inoperable.

The basic idea is that we identify the controlled operations and their authorization requirements, then we verify that the authorization hooks mediate those controlled operations properly. First, we need to identify the representative controlled operations in the kernel. Second, because the controlled operations are at a lower level than the policy operations (i.e., authorization requirements), we need an approach by which the authorization requirements of each controlled operation can be determined. Third, we need to compare the LSM hook authorizations made to the expected authorization requirements. These tasks are complex for in-kernel authorization, so it is obvious that automated support is required.

The mapping of controlled operations to authorization requirements is not necessarily static. For example, a number of the same operations may be executed on a file open for reading as on a file open for writing. Thus, context also is a determining factor in mapping controlled operations to authorization requirements. Our approach must enable context-dependencies to be managed effectively, such that the expected relationships between controlled operations and authorization requirements can be tested.

2.3 Related Work

Recently, static analysis to verify security properties has shown promise. First, existing program analysis tools have been used to find common security errors, such as buffer overflows and `printf` vulnerabilities [Ball et al. 2003; Das et al. 2002; Larochelle and Evans 2001; Shankar et al. 2001; Wagner et al. 2000].

Second, specialized tools have been developed for finding security vulnerabilities, such as xgcc [Engler et al. 2000], ITS4/RATS [Viega et al. 2000], MOPS [Chen and Wagner 2002], MAGIC [Chaki et al. 2003], and so on [Ganapathy et al. 2003].

Static analysis tools are based on formal properties of programming languages, so they can be used for complete analysis (i.e., no false negatives). However, static program verification is computationally expensive, so simplifications are often made in the analysis models. These simplifications can result in more conservative analyses (i.e., more false positives) or abstraction of certain properties (i.e., false negatives). Also, static analysis tools can require a significant amount of effort for code annotation, which is necessary to build the desired analysis model.

Specialized analysis tools focus on specific types of bugs. Engler et al. enables extension of GCC, called *xgcc*, to do source analyses, which they refer to as *meta-compilation* [Ashcraft and Engler 2002; Engler et al. 2000; Hallem et al. 2002]. A rule language, called *metal*, is used to express the necessary analysis annotations in a higher-level language. Rather than annotate the code directly, the metal specifications define finite state automata that guide the analysis engine. Since the rules match multiple statements, the amount of annotation effort is reduced. A variety of software bugs, including security vulnerabilities, have been found by this tool [Ashcraft and Engler 2002].

Most of the specialized analysis tools lack completeness (i.e., may result in false negatives), but MOPS specifically aims for ease of specification and completeness of analysis [Chen and Wagner 2002]. Using MOPS, security properties are expressed as finite state automata and programs are represented as pushdown automata. Data flow is not represented, so aliasing and value relationships are ignored. However, for many analyses useful bugs can still be found [Chen et al. 2004], and it is often possible to show that many data flow relationships do not exist via other means [Zhang et al. 2002].

In another effort, we use one program analysis tool, CQual [Foster et al. 1999], in an approach to finding LSM hook placement bugs statically [Zhang et al. 2002]. Using GCC analysis to automate CQual annotation, we can then perform a CQual analysis that verifies that all controlled operations are mediated by at least one LSM hook. In general, we also want to verify that a controlled operation is only run when its required authorization hooks have been checked. CQual provides a type lattice that could be used for defining expected authorizations, although it is conceptually complex to get it right. Further, the context-dependency on the relationship between controlled operations and authorization requirements is beyond what CQual can handle.

A Java static analysis tool, called *JaBA* [Koved et al. 2002], has been used to collect the actual authorizations on controlled operations for Java. For our purposes, this approach has two shortcomings: (1) it does not analyze the C code of the Linux kernel and (2) it does not provide guidance about whether the authorizations made were correct. On the first point, we have actually defined a translation from C to JaBA analysis concepts [Zhang et al.], and built a prototype implementation. On the second point, JaBA does provide a context-sensitive control flow graph and a context-sensitive data flow graph that can

be leveraged for any analysis. Thus, we will examine use of these graphs in generating an analysis log in Section 5.

Due to the complexity of using these approaches, we found that run-time data collection assisted us in getting accurate data quickly, so that we could explore possible analysis options. From examining the data collected, we have developed a consistency analysis approach that we describe in this paper that enables us to determine whether the appropriate authorization hooks are checked for controlled operations. Ultimately, the approach is independent of whether we do consistency analysis on data collected at run-time or via a static analysis of the code. In this paper, we examine both means of data collection.

Another related problem is the certification of systems. Historically, the Orange Book [NCSC 1985] was used for guidance in the construction of secure operating systems, but this is now being supplanted by the Common Criteria [ITSEC 1998]. However, the certification task is ad hoc and laborious, and has generally not been successful in improving the security of commonly used operating systems. Gutmann argues in his thesis [Gutmann 2000] that certification approaches, including formal verification tools, are doomed to failure unless they represent concepts at the level of the source code. Gutmann also advocates a combination of static and run-time analyses. The approach that we use differs from certification in the sense that it checks for particular errors rather than providing a top-down assurance that the overall system meets its requirements. An interesting research question is whether a sufficient breadth and depth of such checks could provide a confidence comparable to certification. Unlike certification, such confidence could be maintained as the source code evolves.

3. SOLUTION DESCRIPTION

The key insight we leverage in run-time analysis for the LSM framework is that the LSM authorization hook placement is largely correct, such that cases that are inconsistent with respect to the norm are likely to be indicative of an error. For example, it would be considered unusual if a particular controlled operation has different authorization requirements on different runs of the same system call. While this insight does not guarantee that we find all LSM hook placement bugs (see Section 6), it has enabled us to identify some bugs and has served as a valuable guide for the tool development.

In all of the discussion below, we use the following assumptions. First, we leverage an assumption that the objects in controlled operations are handled in a type safe manner in the kernel. This does not invalidate any of the errors we find, but there could be other errors as well. Second, we assume that accesses to objects of the authorized data types define the set of controlled operations (i.e., the mediation interface). These data types are the ones that correspond to system call concepts (e.g., files, inodes, sockets, skbuffs, ipc message queues, and so on). Access to kernel data is designed to go through these data structures. While we have not explicitly validated this, we have done a more detailed analysis presented elsewhere [Edwards et al. 2001].

3.1 Authorization Consistency

We first define consistency between a controlled operation and a set of authorization requirements.

Definition 1 (*Authorization Consistency*). The relationship between a controlled operation and a set of authorization requirements (i.e., policy operations that are authorized) is *consistent* if whenever the controlled operation is executed authorization hooks associated those authorization requirements are called.

We find that this form of consistency is not absolutely required. Execution of a controlled operation may occur in the context of a different system call, which has different authorizations. Clearly, in this case the authorization requirements met will be different.

Thus, it is necessary to be able to define contexts in which consistency is expected. In general, contexts can be arbitrary, but our experience is that three types of contexts matter: (1) system call; (2) system call with specific inputs (e.g., flags); and (3) a specific set of controlled operations. In the first case, the authorization is for the system call at large (e.g., `fcntl`). Such authorizations apply to all the controlled operations in the system call. In the second case, the authorizations depend on some parameter to the system call, usually a flag (e.g., `open` for `read`). Thus, some system calls come under one context and some under another. In the third case, the appearance of the set of controlled operations, independent of the system call in which they appear requires specific authorizations (e.g., the operations associated with accesses to the `set owner` fields). In these cases, the consistency ignores the system call information.

Definition 2 (*Execution Context*). An *execution context* defines a set of execution paths. An execution context can be defined by (1) a system call (all executions); (2) a system call with particular argument values (or ranges of values); and (3) a set of controlled operations (all paths that include them). Other context definitions are possible.

Our solution must support the description of contexts where we expect consistent authorization. Typically, context-sensitive data flow in static analysis refers to distinguishing between different inputs to the same function. In this case, context sensitivity is much narrower (depends only on the system call) or may ignore large parts of the remaining context (for the controlled operation set). Such analyses require significant amounts of annotation for a static analyzer and may depend on variables outside the understanding of the static analyzer (e.g., user-supplied flags). For example, JaBA completely ignores the values of primitive types, but clearly those can influence analysis.

3.2 Authorization Consistency Levels

An execution context usually consists of many controlled operations, so it is helpful to aggregate controlled operations that are consistent in the same way.

Table I. Authorization Consistency Levels: Names and Effects on Authorizations

Level	Authorizations
System call	All controlled operations in system call
Syscall inputs	All controlled operations in same system call with same inputs
Data type	All controlled operations on objects of the same data type
Object	All controlled operations on the same object
Member	All controlled operations on same data type, accessing same member, with same operation
Function	All same member controlled operations in same function
Intrafunction	Same controlled operation instance
Path	Same execution path to same controlled operation instance

For example, if all the controlled operations in a context have the same authorizations, then we can view consistency relative to the context at large rather than the individual operations.

We find that we can describe the consistency between each controlled operation and the authorization hooks that are called when it is executed in a particular context by a set of discrete values we call *consistency levels*. Further, the consistency levels form a total-order as follows.

Definition 3 (*Consistency Level Total Order*). If two different controlled operations are authorization-consistent for the same value of level i, then they are authorization consistent for any value of level j where $i \geq j$ in the *consistency level total order* (see Table I).

If two different controlled operations are executed on the same object, but they have consistent authorizations, then the values of the member and access for those operations do not affect the consistency. For example, if all controlled operations on a particular object have the same authorization requirements, then it does not matter what the member access is. Table I lists the discrete consistency levels. We refer this group of levels collectively as the *authorization consistency levels*. These levels include various aspects of a controlled operation's execution, including the context under which it was executed (system call, system call inputs, function, location in function, path to controlled operation), the object it was executed upon (data type and object), and the operation performed (member and access).

The consistency levels aggregate controlled operations into a *consistency class* where all the controlled operations have the same authorization hooks called given the current placement.

Definition 4 (*Consistency Classes*). Two different controlled operations belong to the same *consistency class* for an execution context, if they have the same authorization hooks called every time they are executed in that context.

3.3 Authorization Consistency Impact

The classification of controlled operations by their authorization consistency divides the controlled operations into two categories: (1) known anomalies

and (2) consistency classes whose authorization requirements need verification. In the first case, we consider some of the authorization consistency levels to be illegal. We define invariants below for these cases. In the second case, we must determine whether the maximal consistency level for each controlled operation in an execution context indicates acceptable authorization requirements or not. For example, if a controlled operation belongs to a group of controlled operations at an object consistency level, this indicates that all the controlled operations on the object have the same authorizations checked. It is then a manual task to determine if this is correct. However, the number of consistency aggregates indicates a partition the controlled operations into maximal-sized classes with the same authorizations. These classes enable verification of authorization requirements and identification of anomalous classifications.

3.3.1 *Anomalies.* The consistency of authorizations to the levels below the double line in Table I, *intrafunction* and *path*, are always considered to be anomalous. Sensitivities of these types mean that the execution path (path) or location within a function (intrafunction) determines the authorization requirements of a particular controlled operation on the same member.

The following invariant formally expresses our path inconsistency invariant.

Definition 5 (*Path Inconsistency Invariant*).

$$\forall c_1, c_2 \in C,\ \ e_1, e_2 \in E, \quad (c_1 = c_2) \wedge (e_1 = e_2) \rightarrow R(c_1, e_1) = R(c_2, e_2) \qquad (1)$$

This invariant states that the same controlled operation ($c_1 = c_2$) run in the same event ($e_1 = e_2$ defined by the system call and its inputs) must have the same authorization requirements (defined by the function R). That is, the execution path within an event cannot affect a controlled operation's authorization requirements.

Similarly, we define an invariant for intrafunction inconsistency.

Definition 6 (*Intrafunction Inconsistency Invariant*).

$$\forall c_1, c_2 \in C,\ \ e_1, e_2 \in E, \quad (F(c_1) = F(c_2)) \wedge$$
$$(M(c_1) = M(c_2)) \wedge (e_1 = e_2) \rightarrow R(c_1, e_1) = R(c_2, e_2) \qquad (2)$$

In this case, two controlled operations in the same function (computed by the function F) and which make the same member access (computed by the function M) must have the same authorization requirements R.

3.3.2 *Authorization Consistency Classes.* For the other cases, we cannot easily identify them as errors. Instead, we partition the controlled operations into their authorization consistency classes and determine whether their authorization requirements are correct.

The authorization consistency class computation is as follows. For each consistency level starting at the highest (system call), we partition the controlled operations into consistency classes where all controlled operations have the

same value for the consistency level, then we test whether the class also has the same authorizations. If not, then we try the next lower level and partition based on both levels and test again. This approach repeats until we have assigned every controlled operation to a consistency class.

Classifications are defined by consistency levels. For the system call level, all the controlled operations of a system call are in one class. For system call inputs, all controlled operations of the same system call and with the same type of inputs are aggregated (see Section 3.4). For the data type level, the controlled operations are classified by the system call, inputs, and data type of the operation's object. Thus, successively finer partitions are created in each step of the analysis.

A classification succeeds (i.e., is x-consistent where x is the level) if it is the first level in which all the controlled operations in that class have the same authorizations. Note that other classes at the same consistency that have the same authorizations are aggregated to form the maximal-sized classes. Once the classes are created it is a manual process to verify that the authorizations for each class are correct. For the file system, the number of classes is small enough that manual verification is practical.

As an example, consider the `read` system call. File operations are data-type consistent because all controlled operations on file objects are authorized for `read`. Manual verification involves checking that read permission for files is sufficient. Since the read authorization also is intended for the file's inode, we mark the file's inode as authorized for `read` as well. However, after classification, one inode's controlled operation is not authorized. It is on a different object, so inode operations may be object-consistent. This is an operation on the directory inode of the file to determine whether a signal should be sent as a result of a read in this directory.[1]

Other than, when an authorization completely missing, the most common way for identifying an error is to find two classifications (i.e., two aggregates with different authorizations) that perform an important common operation. This situation occurred in `fcntl` where two different classifications (based on different system call inputs) operate on the same `f_owner` field (see Section 4.2.4).

3.4 Necessary Data Collection

By logging system call entry/exits/arguments, function entry/exits, controlled operations (i.e., object, data type, member, and operation), and authorizations, we collect all the necessary values for the consistency levels. All the information can be easily logged, but the identification of meaningful object identifiers and system call input changes need some further analysis.

During execution, objects are referenced via function pointers, but this is not necessarily a sufficient identification of an object. For example, an inode has a persistent identifier (i.e., device, inode number) that is used in authorization. Therefore, for each data type we define a specific approach for computing their

[1]Actually, this object should also be authorized by the `read` LSM hook, so we add it to the set of objects authorized by this hook.

Table II. Log Record Types

Record Type	Data		
Controlled operation	Context ID	Controlled operation ID	OID
Authorization	Context ID	Authorization ID	OID
Function entry	Context ID	Instruction address	
Function exit	Context ID		

object identifiers. These identifiers are used for determining all operations and authorizations on an object.

Across system call instances, we assume objects that are used in the same variable have the same authorization requirements. To simulate this, we use the first controlled operation in which an object appears as an identifier. If two objects are first accessed in the same controlled operation they must be assigned to the same variable (since the variable would be the same in the two controlled operations). However, different execution paths may result in the same variable being used in a different controlled operation first. However, aggregation of classes with the same authorization requirements will merge these cases, so this assumption has proven effective.

The system call arguments change on almost every call, but only a few of the arguments really impact authorizations (e.g., the access flag on `open`). Therefore, we collect the arguments, but only use the arguments that we have found impact authorization requirements to do partitioning. Only a few system calls that we have examined have different authorizations based on their input arguments, such as `open`, `ioctl`, and `fcntl`. Because different authorizations are used based on different inputs, these system calls are more complex, and hence, more prone to errors.

4. RUN-TIME IMPLEMENTATION

Run-time analysis of complete authorization consists of two steps: generation of a kernel execution log and its offline analysis for consistency. This section describes the implementation of the tool that creates the execution log, the implementation of the log-filtering tool used to prepare and display analysis data, and the results of our analysis.

4.1 Collecting Run-Time Information

4.1.1 *Log Contents.* Table II shows the information collected during run-time analysis. Controlled operations are identified by the tuple (*instruction pointer, object type, member, access*). A *controlled operation ID* is assigned to each unique combination. Authorizations are uniquely identified by (*LSM_hook, policy operation*). Like controlled operations, a unique authorization ID is assigned to each. Function entry and exit are recorded as well. The function entry address uniquely identifies the function.

For each controlled operation or authorization performed, the log must include the identity of the object (e.g., inode) involved. *Object identities* (OIDs) are defined per object type, for example, inodes are identified by device ID, inode

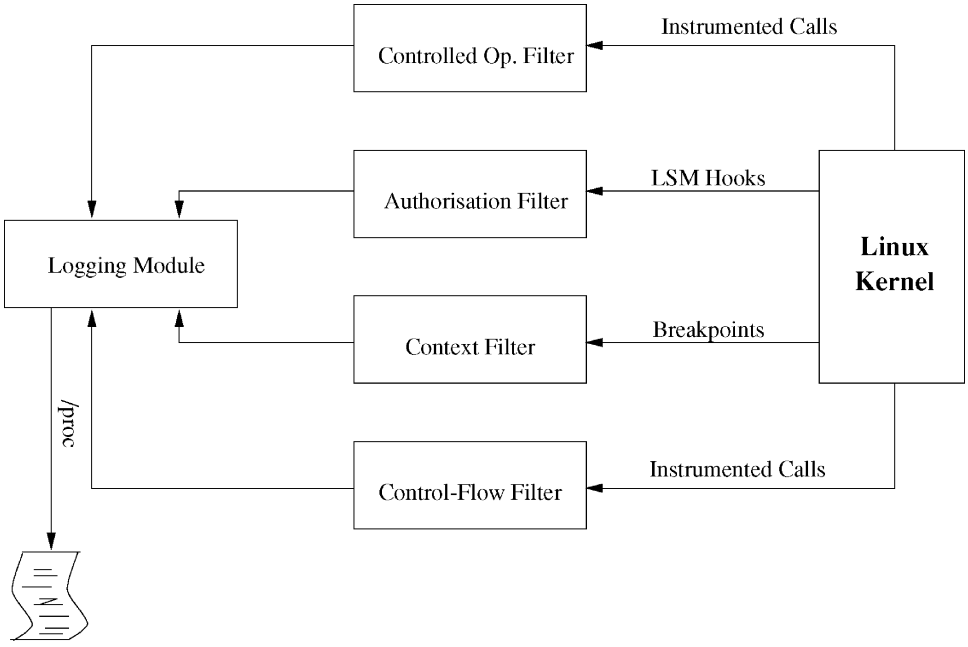

Fig. 3. Implementation architecture.

number, while tasks are identified by process-ID. OIDs are only required to be unique within a context.

We use the concept of a *context* to mean the processing of a kernel event (e.g., a system call). Authorizations are obviously only valid in the context in which they are executed, therefore, the log entries must also include the context of controlled operations and authorizations.

4.1.2 *Collection Overview.* Figure 3 presents an overview of the tool. Creation of the log involves three stages: the required information must be generated, it must be collected, and it must be written to the log.

Information is generated in three different ways. First, authorization information is generated by the LSM hooks. Second, controlled operation details are generated by compiling the kernel with a modified version of GCC that identifies controlled operations, and instruments the kernel with calls to a handler function before all such operations. Control-flow information is also generated by instrumenting the kernel at compile time. Third, context information is generated by placing breakpoints in the kernel. These three methods are discussed in more detail in the following sections.

Fourth, kernel modules are loaded to receive the information shown in Figure 3. These modules perform coarse-grained filtering, and arrange the information into the correct format, before passing the record to the logging module. The logging module assigns a context ID to the incoming records and writes the information into a buffer.

4.1.3 *Authorization Information.* Hooks to log authorization information are already provided by the LSM patch, so little additional implementation is required. The authorization filter is simply an LSM module that adds a log entry for each authorization. These log entries identify the authorization that was performed (e.g., RMDIR_PARENT, RMDIR_TARGET) and the object authorized.

4.1.4 *Controlled Operations.* To log controlled operations, we first have to locate controlled operations in the kernel, and then provide a mechanism for detecting the execution of these operations.

Identifying controlled operations in the kernel requires source analysis. Rather than a direct source-code analysis (which is difficult), we chose to identify controlled operations by analyzing GCC's intermediate tree representation. As Linux depends on GCC extensions, a source-code analysis would require using the GCC parser, therefore making use of the tree it already builds seems logical. To identify controlled operations, we traverse the tree looking for expressions in which members of mediated data types are accessed.[2] When a controlled operation is detected, we insert a call to a function `__controlled_op` that includes the object, type, member, and access, before the statement in which the expression exists. If the expression is the condition statement of a loop, then a call is inserted before the loop and at the end of each iteration. This call contains all the information required to identify the controlled operation and allow the handler to extract the identity of the object.

A couple of accesses cause problems for this approach. First, it is possible to modify a structure member by taking the address of a member, storing it to a pointer, and changing the member via the pointer. Since the initial access is a read into the pointer variable, it is possible that we may miss the subsequent write. Rather than performing more extensive source analysis to identify these cases, we simply detect when aliasing occurs. Second, it is also possible that we miss accesses to controlled data structures when they are cast to a noncontrolled type. This is also detected. Our initial analysis shows that these cases occur in a small number of ways (although for the first, a large number of times), so they can be handled as special cases.

4.1.5 *Control Flow.* Control flow information is generated by compiling the kernel with the `-finstrument-functions` switch provided by GCC-3.0. This option causes the compiler to insert calls to handler functions at the entry and exit of every function. These handler functions then pass the information to the appropriate module.

4.1.6 *Context Information.* As there may be multiple execution contexts in the kernel at anytime, all log entries must contain a context ID, so the analysis can tell which entries relate to one another. Unfortunately, no key is available that will uniquely identify a single execution context, therefore, we

[2]These are COMPONENT_REF nodes where the resultant type of the first operand is a mediated type.

must choose a nonunique key and define an approach to distinguish contexts with the same key.

We chose the base of the current kernel stack as the nonunique key as we need a key that is at least unique among concurrently active executions, and it would seem impossible for this property to be violated for the stack. While it is unique among concurrently active executions, the kernel stack is not unique per-context for three reasons: all system calls from the same process use the same kernel stack, once a process dies its kernel stack may be reallocated to a new process, and interrupts execute with the kernel stack of the process they interrupt. The critical property here is that although the context key is not unique, contexts with the same key are never interleaved. Therefore, by recording the beginning and end of a context (and the associated key), we can unambiguously assign log entries to contexts.

Fortunately, there are only a few points where a context can begin (all located in `entry.S`), and a roughly equal number of places that contexts can end. The exit system call is an exceptional case since it never returns, therefore, the `schedule()` call in `do_exit()` is also identified as a context exit point. Because the number of entry/exit points is manageable, we manually identify their locations for each kernel version and automatically insert them at run-time. To collect this information at run-time, the context filter inserts breakpoint instructions into the (memory-image of the) kernel at all entry and exit points. When a breakpoint is executed, the context filter creates a log entry containing the context key, and whether this is the beginning or end of a context.

4.1.7 *Performance.* We did a simple performance check to determine the performance degradation in the instrumented kernel. On an unmodified Linux kernel, LMBench configured for a "fast benchmark" took 3 min 4 s to run. The instrumented kernel took 3 min 24 s to run the same benchmark for a degradation of slightly over 10%. We believe that this overhead is quite acceptable for such analyses. Recall, that the kernel is instrument for analysis, and the hooks are not required for subsequent use of the kernel.

In this test, as in the results collection described above, we sample 1 out of 20 system calls. The reason for this is to keep the log growth rate lower than the disk throughput rate. Since these benchmarks perform the same system calls many times, we did not notice that we "lost" any security-relevant information. If necessary, a policy for determining when to drop a log entry can be devised.

4.2 Log Analysis

We have also built a tool that enables log analysis for identifying consistencies in authorization requirements as described in Section 3.2. The tool enables the specification of rules for extracting the desired log entries, called *log-filtering rules*, and computes authorization consistency given the extracted entries. We display consistency results in two ways: (1) *authorization graphs* that show the consistency between each authorization and controlled operation and (2) *consistency class lists* that show the aggregation of controlled operations by authorizations and consistency level.

```
# Path-consistent rule for operation at 0xc014f046
1 = (+,id_type,CONTEXT) (+,di_cfm_eax,READ)
2 (D,1) = (+,id_type,CNTL_OP) (+,di_dfm_ip,0xc014f046)
3 (D,1) = (+,id_type,SEC_CHK)

# Member-consistent rule for inode member i_flock read access
1 = (+,id_type,CONTEXT) (+,di_cfm_eax,READ)
2 (D,1) = (+,id_type,CNTL_OP) (+,di_dfm_class,OT_INODE)
(+,di_dfm_member,i_flock) (+,di_dfm_access,OP_READ)
3 (D,1) = (+,id_type,SEC_CHK)

# Input-consistent rule for open for read access, but not path_walk
1 = (+,id_type,CONTEXT) (+,di_cfm_eax,OPEN) (+,co_ecx,RDONLY)
2 (D,1) = (+,id_type,FUNC) (+,di_ffm_ip,path_walk)
3 (D,1)(N,2) = (+,ALL,0,0)
```

Fig. 4. Example authorization consistency filtering rules.

4.2.1 *Log-Filtering Rules.* The log-filtering tool takes an execution log and set of filtering rules as input, and outputs the log entries that match the rules. The rule language is currently rather low level, as we have been concerned more with demonstrating feasibility rather than creating a nice high-level rule language. However, we demonstrate the rule language to give a sense of the types of analyses that are possible.

A rule base is defined by a set of rules that define matching requirements. A rule consists of (1) an index; (2) a dependency specification; and (3) a set of statements. The index identifies the rule within the rule base. The dependency states relationships to other rules by index. We can state that a rule can only match entries that are also matched by another rule, (D, i), where i is the index of the other rule. Also, we can state that a dependency that a rule does not include entries matched by another rule i, as (N, i). Lastly, the statements describe the matching conditions for entries. These are specified by identifying the entry type (id_type), and then matching type-specific levels. Entry types include: events (`CONTEXT`), authorizations (`SEC_CHK`), functions (`FUNC`), and controlled operations (`CNTL_OP`).

Figure 4 shows some example rules. The path-consistent rule finds all authorizations in the context of a `read` system call when a controlled operation at the specified address is run. The first line collects all context entries for a read system call (i.e., the start of the system call). The second line collects all entries of controlled operations at the specified location. The `(D,1)` means that this statement is dependent on statement 1, so only entries within the `read` system call context will be collected. The third line collects all authorizations within the `read` system call context. In this case, each execution of this controlled operation should have the same authorizations or there is a violation of the *path inconsistency invariant* that prohibits a controlled operation from having multiple sets of legal authorizations.

The function-consistent rule collects all authorizations and controlled operations of "read inode member i_flock" within a `read` system call context. The specification of `(D,1)` on the second line means that all controlled operations of this type within a `read` system call will be extracted. If the authorizations

associated with this controlled operation are not the same, then the member access is consistent with its location.

The system call input-consistent rule collects all the log entries in each `open` system call for read-only access. The authorizations of the `open` system call depend on the access for which the file is opened, so `open` is system call input-consistent. Further, we also show a negative filter in this rule that eliminates all entries within the scope of the `path_walk` function. The authorizations for file lookup, including any link traversal, can be separated from those for authorizing the open of this file. Such filtering capabilities enable us to choose our analysis scope flexibly.

The challenge is to write log-filtering rules that express a situation in which a meaningful analysis can be performed. In general, we want the log-filtering rules to describe a situation in which the consistency classes that result meet our expectation. We have found that an optimistic initial assumption of consistency followed by refinement works effectively for designing such rules.

We start by assuming the highest level of consistency, system call consistency for all controlled operations. If all the controlled operations in the system call execution have the same authorizations (i.e., are system call-consistent), then we only have to verify that the authorizations are correct. If not, we examine whether we can write rules at the next level of consistency, the system call inputs. If the inputs have no discernable effect or they do not resolve consistency to our expectation (i.e., all controlled operations do not have the same authorizations), then we go to the next level, data-type consistency, and so on.

Often, effective analysis is possible for lower levels of consistency without changing the rules. We simply need a way to view the aggregates at the same consistency level, such as the mapping of each data type to its authorizations in a data-type-consistent case. We use consistency class lists to view multiple consistency classes (see Section 4.2.3).

4.2.2 *Graphical Log Analysis.* The analysis tool can also generate graphs that enable visual analysis of the filtered data. Using these graphs, it is possible to verify the authorization consistencies by inspection, as we will describe below. An *authorization graph* consists of two sets of nodes in a filtered log: (1) the controlled operations and (2) the authorizations made. Edges are drawn from each controlled operation to the authorizations that have been satisfied when it is run. There are two types of edges: (1) *always* edges mean that the associated authorization is satisfied every time the controlled operation is run and (2) *sometimes* edges mean that the associated authorization is satisfied at least once when the controlled operation is run.

An always edge (as well as the lack of an edge) means that the authorization is not consistent with lower-level levels. A sometimes edge indicates an inconsistency. The lack of an edge where an edge would be expected would indicate a missing authorization.

Figure 5 shows an example authorization graph. The example graph is displayed using the dotty graph visualization tool [Koutsofios and North]. In this case, the authorization graph shows the controlled operations and the authorizations for two types of `fcntl` calls: (1) `fcntl(fd, F_SETOWN, pid_owner)` and

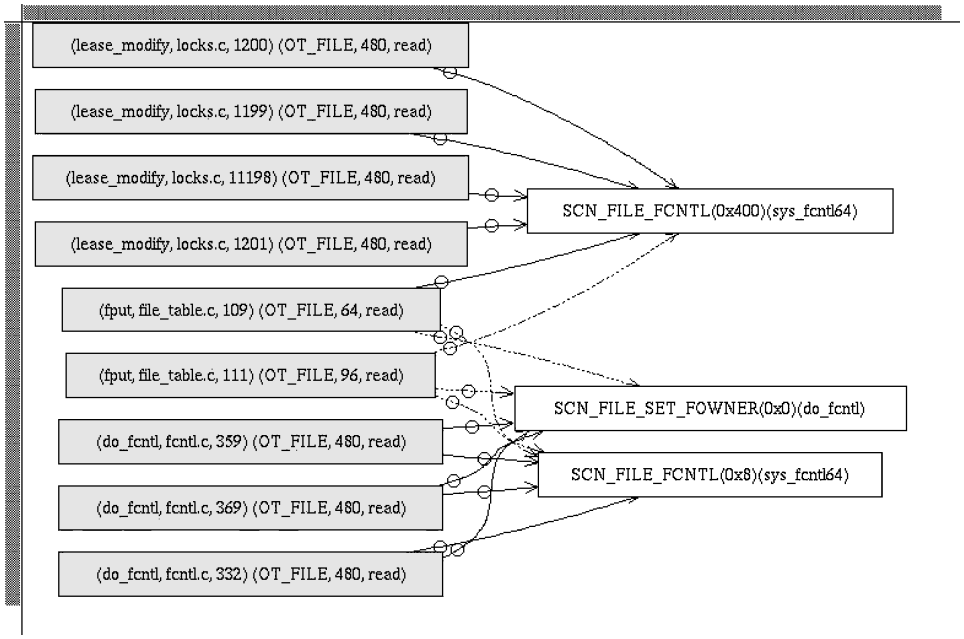

Fig. 5. Authorization graph for `fcntl` calls for `F_SETLEASE` (controlled operations in `lease_modify` and `fput`) and `F_SETOWN` (controlled operations in `do_fcntl` and `put`). When command is `F_SETOWN` both `FCNTL` and `SET_OWNER` are authorized, but only `FCNTL` is authorized for `F_SETLEASE`.

(2) `fcntl(fd, F_SETLEASE,F_UNLCK)`. The controlled operation nodes include location (function name, file name, line number) and operation (data type, member offset, operation type) information. The authorization nodes include the authorization, command, and function containing the authorization. *Always* edges are indicated by a solid line and *sometimes* edges are indicated by a dashed line. If no edge exists between a controlled operation and an authorization, then that authorization is never performed for that operation.

By visually analyzing this graph we can identify whether the invariants described in Section 3.3 hold for the current graph or not. In this case, the *sometimes* relation between `fput` and its authorizations may indicate a problem. Also, the fact that different sets of authorizations are made for the same field (member offset 480 which happens to be `f_owner`) may be indicative of a problem. Manual investigation is then required to identify whether any inconsistency is due to an error or a legitimate consistency.

4.2.3 *Consistency Class Lists.* The consistency class lists show partitions of the controlled operations by consistency level and the authorization requirements at those levels. This partition is computed using the algorithm described in Section 3.3. The consistency class lists provide a different view than the authorization graphs of the same authorization results. Whereas an authorization graph shows the relationship between each individual controlled operation and authorization, the consistency class lists show collections of controlled operations with the same authorization requirements. The consistency class lists

```
DFN d 0 FILE f_dentry -1
DFN d 0 FILE f_dentry 1
DFN d 0 FILE f_vfsmnt -1
DFN d 0 FILE f_op -1
...
SFN(ALWAYS) d 0 FILE_READ
---------------
DFN d 1 SUPERBLOCK s_blocksize -1
DFN d 1 SUPERBLOCK s_type -1
...
DFN d 1 TASK state -1
DFN d 1 TASK state 0
DFN d 1 TASK flags -1
...
SFN() NONE
---------------
DFN o 0 INODE i_blocks -1
DFN o 0 INODE i_blocks 1
DFN o 0 INODE i_version -1
...
SFN(ALWAYS) o 0 FILE_READ
---------------
DFN o 1 INODE i_dnotify_mask -1
SFN() NONE
---------------
```

Fig. 6. Consistency class list for `read` system call with the following fields: (1) entry type (DFN or SFN); (2) consistency (`d` for data type and `o` for object); (3) class number; (4) data type; (5) member; (6) access identifier.

makes more obvious the number of different authorization cases that exist in the data. Also, the consistency class lists are easier to use in regression testing since they are textual [Jaeger et al. 2002].

Figure 6 shows the partition of controlled operations into consistency classes for the `read` system call. This partition is used as the example in Section 3.3. As described there, the consistency class list shows two classes that are consistent at the data type level: one for tasks and superblocks with no authorizations and one for files with read authorization. Then, the consistency class list has two classes that are object-consistent: one for the inode that is read authorized and one for its directory that has no authorizations. Ultimately, we expect to annotate current task, file's directory, and file's superblock as read authorized which will result in all controlled operations having the same authorization (i.e., being system call-consistent).

Most of our experience is with the file system although we have also examined task authorizations. Most objects have either one or no authorizations, so the consistency class lists are not too complex. The system call `unlink` is one of the few where an object has multiple authorizations. Using consistency class lists it is easy to see that the directory inode has three authorizations (exec, write, unlink_dir) and the inode being removed has one (unlink_file) because they are object-consistent and placed in different classes. Thus, for the file system and the task operations we have examined, authorization graphs and consistency class lists have been sufficient to verify authorizations.

```
# fcntl for F_SETOWN with just the field f_owner
1 = (+,id_type,CONTEXT) (+,di_cfm_eax,fcntl) (+,co_ecx,F_SETOWN)
2 (D,1) = (+,id_type,SEC_CHK)
3 (D,1) = (+,id_type,CNTL_OP) (+,di_dfm_member,f_owner)
# fcntl for F_SETLEASE with just the field f_owner
4 = (+,id_type,CONTEXT) (+,di_cfm_eax,fcntl)
(+,co_ecx,F_SETLEASE) (+,co_edx,F_UNLCK)
5 (D,4) = (+,id_type,SEC_CHK)
6 (D,4) = (+,id_type,CNTL_OP) (+,di_dfm_member,f_owner)
```

Fig. 7. Rules for finding the `f_owner` anomaly.

4.2.4 *Sample Analysis.* We briefly demonstrate a sample analysis for an anomaly that we found. While the approach to finding anomalies was developed concurrently to actually finding anomalies, we used roughly the same approach as described although some of it was not automated. This anomaly occurs in the `fcntl` system call. The consistency class list for `fcntl` shows that its authorizations are system call input-consistent. The values of the `cmd` and `arg` parameters to `fcntl` can change the authorizations that are required. We use authorization graphs to look at the authorizations under the different inputs since it is easier to see coarse-grained problems—lots of *sometimes* edges occur.

Figure 7 contains two sets of rules: (1) one which collects all authorizations and controlled operations of the file structure field `f_owner` in a `fcntl(fd, F_SETOWN, pid_owner)` system call, and (2) one which collects all authorizations and controlled operations on the field `f_owner` in a `fcntl(fd, F_SETLEASE, F_UNLCK)` system call. Note that this is same rule (less the `fput` controlled operations) used to generate the graph in Figure 5.

In Figure 5, we see that some of the controlled operations are authorized for the `fcntl` and `set_fowner` authorizations and some are only authorized for `fcntl`. This is despite the fact that the controlled operations access the same field, `f_owner` (offset 480). Given this anomaly, we examined the kernel source to determine whether an exploit of this anomaly is possible. We discuss the results of this analysis in the next section.

4.3 Results

We applied the December 10, 2001 LSM patch to the Linux 2.4.16 source and compiled the kernel using our modified version of GCC-3.0.[3] To create an execution log to analyze, we executed in parallel three instances of LMBench, the SAINT vulnerability tool (`www.wwdsi.com/saint/`), a kernel compile, and some test programs that we wrote as we became suspicious of anomalies. Since the effectiveness of run-time analysis depends on running enough code, the development of benchmarks that cover enough of the interesting paths must be developed. For example, LMBench only runs about 20% of the kernel code.

[3]Keeping up with kernel version is not a great deal of work. We have the system running on Linux 2.4.18 now, and the only thing we had to do was update our authorization filter to the current LSM interface. However, the LSM interface is being redesigned significantly as part of its inclusion in the main Linux kernel, so we have not yet updated for this redesign. It should not take more than one day to update.

We have instrumented the kernel to collect controlled operations on the major kernel data structures: files, inodes, superblocks, tasks, sockets, and skbuffs. So far, we have only done a detailed analysis on the file system authorizations, and an initial analysis on task authorizations. Since the file system is fairly well understood, we did not expect a large number of anomalies, but we found some nonetheless.

—**Member-consistent (multiple system calls):** `setgroups16` is one of several backwards ABI-compatible 16-bit task operations, such as `setuid16` and `setchown16`. These operations usually convert their 16-bit values to 32-bit values and call the current versions that do contain authorizations. However, since `setgroups16` sets an array, it is easier not to convert the array, so the current version (that contains a hook) is not called. We found that there is no authorization hook in the function `setgroups16`, so we can reset the task's group set without authorization. Note that there is no `setgroups16` call in the current version of libc, so we had to write an assembler program to perform this exploit.

—**Member-consistent (single system call):** The `f_owner.pid` member of `struct file` tells the kernel which process to send signals to regarding IO on this file. Setting this field is authorized by `file_ops->set_fowner` if the user tries to set it directly via `fcntl(fd, F_SETOWN, pid_owner)`. However, if a user removes a lease from a file via `fcntl( fd, F_SETLEASE, F_UNLCK)`, the owner is set to zero without the authorization being performed. Furthermore, a process can set the owner of a Universal TUN device (drivers/net/tun.c) to itself without the authorization being performed. To achieve this, the process calls `ioctl(fd, F_SETFL, FASYNC)` on an open, attached, TUN device.

—**Member-consistent (single system call):** During our investigation of the consistency of `filp.f_owner` described above, we found that access to `filp.f_owner.signum` (the signal that should be sent upon IO completion) can be set without the authorization via `fcntl(fd, F_SETSIG, sig)`.

—**System call-consistent (multiple system calls):** A `read` authorization is performed at the beginning of every read system call. This authorization is required since the authorization performed when the file was originally opened may no longer be valid, due to the process changing its security levels, the file changing its security levels, the file being used by a new process, or a change in the security policy. This authorization, however, is not performed during a page-fault on a memory-mapped file. Therefore, once a process has memory-mapped a file it can continue to read the file regardless of changes to security levels or security policy.

We engaged in a discussion with the LSM community that resulted in a patch to all the anomalies, except the one for reading memory-mapped files. The community decided that a file that requires read authorization must not be memory-mapped. We are encouraged that we have been able to help find and fix hook placement problems.

5. EXAMINING STATIC ANALYSIS

We have now implemented a static analysis prototype using the JaBA static analysis tool that collects logs of relevant execution paths in the Linux kernel. For static analysis, full kernel execution path coverage can be more directly assessed than for run-time analysis. Unfortunately, the number of kernel execution paths is exponential in general, so the model must be designed to remove as many irrelevant paths as possible and be optimized so as many paths can be assessed as possible. The statically generated logs can be input into the analysis tools described in Section 4.2 as a run-time generated log would be.

5.1 JaBA Analysis Tool

JaBA is a flow-sensitive, context-sensitive static analysis tool with pointer-based data flow analysis [Koved et al. 2002]. JaBA has base functions that process Java class bytecodes to build intraprocedural basic block graphs, interprocedural control flow graphs, and data flow graphs for variables and object fields. We build further static analyses for the log collection using this base analysis information as input. These analyses are procedural, as opposed to the declarative analysis languages, such as MC [Engler et al. 2000]. We then process these logs as described in Section 4.2.

Context-sensitive control flow in JaBA means that different paths into the same function can be distinguished from one another, such that different input objects into the same function can be distinguished and their relationships to authorizations can be tracked accurately.

Context-sensitive data flow in JaBA tracks data flow from the allocation sites in the program. For the Linux kernel, the same allocation sites are used for most objects (e.g., files and inodes), so in a context-insensitive data flow, the same objects would be used in all operations of that type. JaBA enables the definition of context-sensitivity by distinguishing objects by the call site into the function that creates the object (or arbitrary levels). The more objects created, the more expensive the analysis becomes however.

Since the Linux kernel is obviously not in Java, we have written a transformation from C to Java bytecodes [Zhang et al.]. The transformation preserves only the semantics of the Linux kernel that are relevant to the JaBA analyses. For example, pointers to C structures are translated to arrays of size 1 containing a Java object of the commensurate type. Also, nontype safe pointer operations are translated to the object `wildPtr`.

5.2 Log Generation Process

The JaBA log analysis process uses two inputs: (1) a root control flow graph node, usually a function for a system call (e.g., `sys_open`) and (2) the data types of the structures whose controlled operations are being analyzed. All JaBA analysis is restricted to the functions (i.e., CFG nodes) that are reachable from the root node. The log generation returns a log of the form collected by the run-time data collection tool.

The log generation process using JaBA is as follows:

(1) Dump id to name mapping for functions, authorizations, and controlled operations for use by log analysis tool.
(2) Collect all controlled operations on objects of the specified data types. Recall that these data types are specified as input to the analysis process and the scope of the collection is limited by the functions reachable from the root.
(3) Collect all authorizations on objects of the specified data types.
(4) Define a consistency context for each path, currently defined to be each individual object instance used in a system call. This context may also include conditional branch restrictions to emulate switches due to system call flag values.
(5) Identify nodes with paths that lead to either a controlled operation or an authorization. All other nodes are irrelevant to the analysis.
(6) Generate all relevant intraprocedural paths (see below).
(7) Generate interprocedural paths from sequence of intraprocedural paths.
(8) Repeat from step 4 until no new contexts to examine.

The first step enables the log analysis tool to properly display the results from the JaBA-generated logs by providing a mapping from numerical representation (e.g., function CFG node number) to names (e.g., function names).

The second step collects all controlled operations for the data types of interest as is done in the run-time case. In this proof-of-concept, we only collect write accesses thus far. The collection of read accesses is necessary to find some errors (e.g., read from `f_owner` field to update its members), so extension to read accesses will be necessary. While the addition of more accesses will only increase the complexity of the analysis, we do not think the additional complexity will be significant as discussed in Section 6.

The third step collects the authorizations for the data types of interest. In the run-time system, we had to manually list all the objects that were thought to have been authorized by each LSM hook. Here, we are hoping to use data flow to more accurately describe the objects that are authorized as discussed in Section 6. However, at present, only the direct authorizations are collected. We are defining a mechanism to infer authorized objects from an authorization, but this step is currently less extensive than in the run-time analysis.

The fourth step defines a set of analysis contexts. In the run-time tool, the execution of a system call defines a context. In the static analysis, a system call defines many paths, so we need to identify the path contexts that are worthy of collection. We have identified a number of ways to reduce the number of intraprocedural paths worthy of consideration (see Section 5.3), but we still found that further reduction is necessary for a timely analysis. We follow the lead of *xgcc* project where analysis automata processing is variable-dependent [Hallem et al. 2002].

Since JaBA tracks objects rather than variables, objects define contexts. That is, for each system call, one object is identified and only operations and authorizations on that object define a context. After path analysis for that object is

complete another object in the system call is selected and all relevant contexts (i.e., paths) for that object are logged. This repeats until all relevant objects (i.e., objects of the target data type) in each system call are analyzed and their logs are collected.

The fifth step takes the context description and determines the CFG nodes that are worthy of analysis. A node is worthy of analysis if its removal and thus the removal of any of its descendants would result in the loss of a controlled operation or authorization log entry. Thus, functions that call any function that has a controlled operation or authorization relative to the current context object are worthy of logging.

The sixth step creates all relevant intraprocedural paths for nodes worthy of analysis. This is the key step in the analysis, and it is discussed in detail in Section 5.3.

The seventh step combines intraprocedural paths into individual logs. Logically, it enumerates all combinations of intraprocedural logs in execution order. In actuality, log generation is optimized to build new logs from the function that provides the new intraprocedural path for the next combination, rather than building each path from scratch. Logs are built by collecting a sequence of intraprocedural paths for those functions that would be called in those paths. On the next pass, the last function in the log that has multiple intraprocedural paths is chosen to define the next combination. The log up to that point remains the same, so log generation is done only from the start of the new path.

The log analysis described in Section 4.2 is performed as before on the logs generated in step (7). The only differences in log entries are: (1) JaBA object identifiers are used for objects rather than object-specific identifiers as defined in the run-time and (2) primitive arguments are not captured by JaBA, so system call flags must be represented by a conditional branch identifier which we store in the system call arguments locations for the first four conditionals at present.

5.3 Intraprocedural Analysis

Intraprocedural analysis defines the number of relevant paths per node. Since intraprocedural path combination is exponential in general, all reasonable steps should be taken to keep the number of paths as minimal as possible. We perform both node-level and conditional-level analyses to try to eliminate redundant paths. As a result, only one function in the Linux file system requires more than three paths and most have only one path.

The main thing we are looking for are the relationships between controlled operations and authorizations. Thus, if all the paths in a node result in the same relationships, then all the paths in the node can be combined into one. Because the number of authorizations is much smaller than the number of controlled operations, we take an authorization-centric view: If all the paths in a node have the same authorizations (i.e., none lead to a new authorization for the context object), then all the controlled operations can be combined into one path. Since the authorizations typically occur before the controlled operations, this optimization is quite effective.

Different paths in a function are created by its conditional statements. A condition states that either the one of two or more paths can be taken, so each choice forms a path. If the choice of paths does not impact the relationship between the controlled operation and the authorizations, then these conditional paths need not be considered independently. We refer to this process as *merging*. In most functions, this results in a significant reduction in the number of paths.

For intraprocedural analysis we perform the following steps:

(1) Sort the nodes in topological order with CFG leaves first.
(2) Determine if the node calls any nodes that have authorizations. If not, compress the controlled operations into one path.
(3) Collect node's basic blocks into a path. If a conditional statement is found, determine if this conditional can be "merged." If not, choose the first path and continue. Push the conditional on a stack for subsequent path generation.
(4) Store function call locations in path for interprocedural path generation.
(5) When an intraprocedural path is complete, rewind the conditional stack until a conditional is found that has branches left to examine. Generate the next intraprocedural log from this location.

The only function in the Linux virtual file system that caused difficulties was `path_walk`, which has 28 relevant paths. First, this is a very long and complex function with many possible paths. However, we believe that this function calls subroutines that actually refer to semantically different objects from our perspective, so the JaBA data flow analysis does not capture the right semantics yet. Many of the subroutines should not really be relevant to the analysis. Regardless, the generation of `path_walk` paths take 3 min, and the generation of interprocedural paths for system calls that use `path_walk` (there are plenty), still quite on the order of seconds since most other nodes have 1 or 2 paths.

The overall processing time for the system calls over the Linux virtual file system takes approximately 1 h, which is actually analogous to the time it takes to put together a run-time analysis. However, we are optimistic that static analysis performance can be improved significantly. Currently, the way our JaBA system is architected we must recompute `path_walk` for each system call in which it is used. Removing this redundant calculation will reduce the analysis time by slightly more than 50%. Also, we can improve performance further by starting the analysis in the node an object is first seen rather than at the system call.

5.4 Static Consistency Results

We aim to use the JaBA consistency analysis to find both the `file_ops->set_fowner` error described in Section 4.3 and the TOCTTOU vulnerability [Bishop and Dilger 1996] that we previously found using the static analysis tool CQUAL [Zhang et al. 2002]. Both can be categorized as consistency errors, but the latter is difficult to find with a run-time analysis because it requires an active successful attack to cause the inconsistency. If we already knew the vulnerability, we would not need the tool.

Recall that the `file_ops->set_fowner` error was caused because the controlled operations on that field (read operations in this case) were authorized by a `set_fowner` for all functions except `lease_modify`. The static log includes entries for `fcntl_setlease`, which indicate that in some logs `set_fowner` is authorized and in some logs it is not. Interestingly, the log does not collect log entries for `lease_modify` because it is always called with the same authorization, only `fcntl`. Because the relationship is fixed, we need not consider the paths within the function. However, we do need to consider that a particular controlled operation may be used that may identify a consistency problem, as is the case here. In this case, we want to know that the log may call `lease_modify` and other functions and execute their controlled operations. Since the relationships between the controlled operations and authorizations are static, we can aggregate all the controlled operations (i.e., merge them into one aggregate path). Doing this would enable the Vali analysis tools to identify the lack of an authorization in `lease_modify`.

The TOCTTOU vulnerability found in Zhang et al. [2002] occurred because the file pointer is authorized in `sys_fcntl`, but a new file pointer is extracted based on the user-provided file descriptor in `fcntl_getlk`. Since the user can control the mapping between the file descriptor and the file it references, a race condition results that an attacker can leverage to perform unauthorized `fcntl` operations. JaBA provides a data flow graph that tracks the possible values of variables and fields in the program. In this case, we need to distinguish the file pointer variables based on their source (i.e., `fget` from the file descriptor). The static logs generated indicate that the file pointer object is `sys_fcntl` was authorized in all cases, while a different file pointer object is used in `fcntl_getlk` that is never authorized.

Thus, our analysis tool built on JaBA enables the construction of analyses that can implement a more general consistency analysis than the run-time tool that can cover our previous static analyses as well.

6. DISCUSSION

In this section, we briefly examine three issues in the use of the LSM consistency analysis: (1) static analysis effectiveness; (2) verification of LSM hook placement; and (3) use for regression testing.

6.1 Static Analysis Issues

In general, we find that we can use the described consistency analysis on log data collected via either run-time or static methods. The current proof-of-concept static analysis log collection takes about the same amount of time as building and executing the run-time log collection. The same log analysis can be performed using either tool.

Static analysis has the advantages of path coverage and accuracy of analysis. First, it is difficult to find benchmarks that execute all relevant paths. Basically, a static analysis of the code is necessary to write the necessary benchmarks to execute the proper paths. Second, the previous CQual static analysis demonstrated that TOCTTOU bugs could be found that cannot be found

using run-time analysis. Finding these bugs using a run-time analysis requires that there be an active attacker performing the TOCTTOU attack, so we would already have to know that the attack exists.

Effective use of static analysis depends on handling all necessary controlled operations (i.e., adding read accesses) and use of data flow to identify authorized objects. Extending analysis for read accesses will extend the length of the log and could increase the number of relevant paths. The relevant paths depend on new relationships between controlled operations and authorizations, but since the number of authorizations is small for each system call, we do not expect this to be a problem. We can further refine the notion of a context to further restrict the scope of log generation.

An important use for static analysis is to deduce those objects authorized by an LSM hook. The LSM hooks do not describe explicitly the objects that they authorize, so this must be determined. Using the run-time analysis such properties had to be specified manually, but static analysis may enable such properties to be inferred. In general, objects passed to the hook are authorized. Also, objects that are referenced by the authorized objects via immutable fields can also be assumed to be authorized. Whether the fields are immutable depends on whether there is a system call that can obtain an alias to the same object and can modify the field. Object creation calls typically do not qualify because they create a fresh object, so they cannot modify an existing object. While the file–file descriptor relationship can be modified by a system call, the relationship between a file object and its inode is likely not modified once the file object is created. We are examining data flow analyses that enable such properties to be verified.

6.2 Verification

Most analysis work to date aims at bug finding. Although some analysis approaches are complete (i.e., no false negatives, such as MOPS [Chen and Wagner 2002]), these tools have not been used to prove any significant, even minor, security property. With sufficient analysis support, we would like to be able to actually verify system security properties. We examine our status in achieving this goal.

Our analysis enables the verification that a particular set of *mediating operations* (e.g., LSM hook authorizations) are consistently performed when a set of *use operations* (e.g., kernel controlled operations) are executed. If the mediating operations can be identified in advance (i.e., we can identify the authorizations required) and they can be related to the use operations (e.g., by the objects and operations), then this analysis is complete.

In the LSM hook analysis, the authorizations for a controlled operation are not specified in advance, so the consistency analysis is said to determine the required operations. There are three issues that make this assumption difficult: (1) there may be LSM hooks that are just completely missing, so there is no inconsistency issue; (2) the context descriptions may not cover all relevant paths; and (3) the set of objects identified by an LSM hook may not include some objects that are implicitly authorized. In the first case, there is no inconsistency

problem if a hook is missing in all cases. The idea is that the aggregation of the authorization-controlled operation relationship make this verifiable manually. However, there is the possibility for error in the manual verification. In the second case, context specifications must ensure that consistency analysis covers relevant log paths. To achieve this, each controlled operation must belong to a context statement its authorization relationships are consistent and verified against missing hooks. The simplicity of the three types of contexts (i.e., recall system call, system call flag, and operation set) makes this coverage fairly easy for LSM hooks. In the third case, the LSM hooks do not explicitly state the objects that are authorized. As discussed in the previous section, we aim to use JaBA data flow analyses to verify this property.

Thus, if we can ensure complete coverage of all kernel paths that impact consistency, associate all mediated objects with their authorizations, and reliably verify authorization requirements then we can perform effective verification. We expect that we can verify properties such as object labeling (label before use), object initialization, audit, and object reuse. For example, audit operations should be paired with security decisions. These are easily identified, so this verification should be straightforward although inferring the effected objects is still an issue. For labeling, we can identify labeling operations and determine whether they are run on all objects prior to security checks. However, objects can be relabeled. Determining whether relabeling occurs where necessary requires defining the requirements of relabeling. That is, the mediating operations that are indicative of a relabel and the controlled operations indicative of use of a relabeled object have to be determined. Thus, the verification of initial labeling may be possible, but the verification of correct relabeling appears to be much more difficult.

6.3 Regression

Once a security property is verified, it is necessary to ensure that this property is maintained as the system is updated. This process is generally referred to as regression testing. Since updates can be made by lots of people in an open source system, such as Linux, it is necessary for most regression to be much simpler than the original verification. In particular, we would like to maximize the possibility that a simple change will result in no manual examination of regression data.

For LSM hook verification, the use of consistency classes provides some resilience against change. As long as the consistency class list does not change, then the output is basically the same. Since changes in consistency are indicative of placement issues, consistent consistency in regression testing indicates effective placement. In order for this assumption to hold, we also must account for the fact that a necessary hook may be missing in all cases. Thus, as long as no new controlled operations are made, this regression approach works. For the addition of new controlled operations, more needs to be done.

Examination of the other contexts in which the controlled operation appears whether it is benign or a potential problem. In general, the larger the number of different sets of authorizations to which a controlled operation is associated

in consistency lists is one measure of its independence from particular authorizations. This case would be considered *low* for requiring a new LSM hook. On the other hand, if a controlled operation is part of a set of controlled operations that define a context, then the likelihood of the need for an LSM hook is much higher. We can aggregate new controlled operations based on these kinds of criteria to indicate severity of need for deeper review.

7. CONCLUSIONS

In this paper, we presented a consistency analysis for assisting the Linux community in verifying the correctness of the LSM framework. The LSM framework consists of a set of authorization hooks placed inside the kernel, so it is more difficult to identify the complete mediation points. We leveraged the fact that most of the LSM hooks are properly placed to identify misplaced hooks. We used structure member operations on major kernel data structures as the mediation interface and collected the authorizations on these operations. By analyzing the output of a run-time logging tool, we identified the operations whose authorizations were inconsistent. We have analyzed the file system and some task operations and found some anomalies that could have been exploited. Working with the LSM community, these problems have since been fixed. For example, we found that some variants of `fcntl` enabled operations to be performed that were authorized in other cases.

Ultimately, we found that consistency analysis is useful for verifying systems where a inconsistencies from the norm are likely to be errors. However, run-time log collection has limitations in path coverage and can miss some important errors, such as TOCTTOU, that depend on unusual input data or contexts. We have found that this analysis approach can also be applied to execution logs collected via a static analysis. We demonstrate the use of the JaBA analysis tool to collect such logs. Further, we discuss improvements to the overall analysis that can be made using the data flow analysis of JaBA. We are actively pursuing complete, easy-to-use verification for the LSM authorization hooks and examining verification of other security properties.

REFERENCES

ANDERSON, J. P. 1972. *Computer Security Technology Planning Study*. Tech. Rep. ESD-TR-73-51, Air Force Electronic Systems Division.

ASHCRAFT, K. AND ENGLER, D. 2002. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the IEEE Symposium on Security and Privacy*.

BALL, T., COOK, B., LEVIN, V., AND RAJAMANI, S. K. 2003. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Methods*, Lecture Notes in Computer Science, vol. 2999. 1–20. Also appears at MSR-TR-2004-8.

DAS, M., LERNER, S., AND SEIGLE, M. 2002. ESP: Path-sensitive program verification in polynomial time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*.

BISHOP, M. AND DILGER, M. 1996. Checking for race conditions in file accesses. *Computing Systems*, *9*, 2, 131–152.

CHEN, H. AND WAGNER, D. 2002. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the 9th Conference on Computer and Communications Security*.

CHEN, H., DEAN, D., AND WAGNER, D. 2004. Model checking one million lines of C code. In *Proceedings of Network and Distributed System Security Symposium (NDSS 2004)*.

CHAKI, S., CLARKE, E., GROCE, A., JHA, S., AND VEITH, H. 2003. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*. 385–395.

EDWARDS, A., JAEGER, T., AND ZHANG, X. 2001. *Runtime verification of Authorization Hook Placement for the Linux Security Modules Framework*. Tech. Rep. RC22254, IBM Research.

ENGLER, D., CHELF, B., CHOU, A., AND HALLEM, S. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operation System Design and Implementation (OSDI)*.

HALLEM, S., CHELF, B., XIE, Y., AND ENGLER, D. 2002. A system and language for building system-specific, static analyses (appeared in PLDI 2002). In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*.

FOSTER, J., FAHNDRICH, M., AND AIKEN, A. 1999. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*. 192–203.

GANAPATHY, V., JHA, S., CHANDLER, D., MELSKI, D., AND VITEK, D. 2003. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*.

GUTMANN, P. 2000. *The Design and Verification of a Cryptographic Security Architecture*. Submitted thesis. Available at `www.cs.auckland.ac.nz/pgut001/pubs/thesis.html`.

ITSEC. 1998. *Common Criteria for Information Security Technology Evaluation*. ITSEC. Available at `www.commoncriteria.org`.

JAEGER, T., ZHANG, X., AND EDWARDS, A. 2002. Maintaining the correct of the Linux Security Modules framework. In *Proceedings of the 2002 Ottawa Linux Symposium*. 223–241.

LARSON, E. AND AUSTIN, T. 2003. High coverage detection of input-related security faults. In *Proceedings of the 12th USENIX Security Symposium*.

KOUTSOFIOS, E. AND NORTH, S. Drawing graphs with Dot. Available at `http://www.research.att.com/sw/tools/graphviz/`.

KOVED, L., PISTOIA, M., AND KERSCHENBAUM, A. 2002. Access rights analysis for Java. In *Proceedings of 17th Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

LAROCHELLE, D. AND EVANS, D. 2001. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Tenth USENIX Security Symposium*. 177–190.

NCSC. 1985. *Trusted Computer Security Evaluation Criteria*. National Computer Security Center. DoD 5200.28-STD, also known as the Orange Book.

SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. 2001. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the Tenth USENIX Security Symposium*. 201–216.

VIEGA, J., BLOCH, J., KOHNO, Y., AND MCGRAW, G. 2000. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of 2000 Annual Security Applications Conference*.

WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium (NDSS 2000)*.

ZHANG, X., EDWARDS, A., AND JAEGER, T. 2002. Using CQual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*.

ZHANG, X., MARCEAU, G., JAEGER, T., AND KOVED, L. Translation of C programs for static analysis by the JaBA framework. Internal draft document to become an IBM technical report.

Received July 2003; revised March 2004; accepted March 2004