CERIAS Tech Report 2007-68

Run-Time Label Propagation for Forensic Audit Data

by Florian Buchholz and Eugene H. Spafford

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Run-time Label Propagation for Forensic Audit Data

James Madison University Infosec Techreport
Department of Computer Science

JMU-INFOSEC-TR-2006-001

Florian Buchholz and Eugene H. Spafford

May 2006

Run-time Label Propagation for Forensic Audit Data

Florian Buchholz[†] and Eugene H. Spafford[‡]

[†] *Department of Computer Science, James Madison University, Harrisonburg, VA 22807*

[‡] *CERIAS and Department of Computer Science, Purdue University, West Lafayette, IN 47907*

buchhofp@jmu.edu spaf@cerias.purdue.edu

Abstract

It is desirable to be able to gather more forensically valuable audit data from computing systems than is currently done or possible. This is useful for the reconstruction of events that took place on the system for the purpose of digital forensic investigations. In this paper, we propose a mechanism that allows arbitrary meta-information bound to principals on a system to be propagated based on causality influenced by information flow. We further discuss how to implement such a mechanism for the FreeBSD operating system and present a proof-of-concept implementation that has little overhead compared to the system without label propagation.

1 Introduction

Security mechanisms on computing systems, such as intrusion detection systems, access control, and audit facilities rely on the information that is available on the system on which they are deployed. The information that is available on computing systems about the events that occur, however, did not evolve from the need for good security data but rather from the need to manage the available shared resources of the system among its users. If the need arises to investigate an incident, as part of a forensic investigation or incident response, the information available to an investigator is often further reduced, as not everything is recorded on a permanent basis. This makes it difficult to draw sound conclusions from the available information. Most of the effort to date in the digital forensics community has been in the retrieval and analysis of existing information from computing systems. Little has been done to increase the quantity and quality of the forensic information on computing systems.

An operating system's main function is to administer the limited available resources to the programs that request them. Thus, much of the information a system

keeps about its processes and objects is related directly to the task of administering those resources. A large part of this information is kept for reasons of access control. Other important security concerns do not play a prominent role. While processes usually carry a user identifier, it may be unclear whether this user is truly responsible for the actions the process performs. Furthermore, there is no notion at all about location, or origin, of a system's processes and objects. From where was a session initiated? Where did a file come from?

Third party extensions exist that add more information for the purpose of access control [41] but also for detecting policy violations [25, 48], adding a sense of location [41, 14] or more general security audit mechanisms [28, 40]. Logging facilities such as `syslog` for UNIX and the Event Viewer mechanism for the Windows platform may record extra information about events as they occur. However, most of these extensions lie outside the system itself, which means that they may be subject to tampering. Furthermore, separate event logs need to be correlated to establish any causal relationship between events. This is at best a tedious task, but may even turn out to be impossible in certain situations because not enough information is recorded or propagated.

When adding new audit information to a system, the goal should be to preserve event relationships. From a security perspective, one wants to determine answers to questions such as "who is responsible for events?" or "where are the entities that caused the events located in the world (or network)?" To achieve this, it is not sufficient to simply have one user identifier per process or file, or introduce a location field. This is because a principal acting within a system is influenced by other principals or by the contents of objects such as files. This causality is governed by the information flow among principals and objects on the system. A principal communicating with another principal may influence/force/ask/trick/signal the latter into performing some sort of action. The same is true for the content of an object that a principal accesses. In this paper we present a methodology that allows the system to bind arbitrary information in the form of a label to its principals. Labels are then propagated to other principals and objects on the system as information is exchanged between them. Depending on the nature of such a label, e.g. user identity or location information, valuable audit data can be created on a system. This is especially useful for digital forensics, intrusion detection, network traceback, and access control.

In this paper we present a propagation model and a proof-of-concept implementation, where labels are propagated based on information flow between subjects. This includes cases where one principal (or more) controls the actions of another. Our approach differs from traditional information flow analysis methods in the way that we do not attempt to determine how information actually is exchanged but rather determine who communicates with whom. We will focus on two categories of information that may be desired during a forensic investigation but current operating systems cannot supply. These two categories are *user influence* and *location information* (host causality [7]). The need for this kind of information and other categories is discussed

in previous work [8].

The remainder of this paper is organized as follows: Section 2 defines the problem space and addresses related work. Section 3 introduces our propagation model and discusses its properties. In Section 4 we discuss aspects for implementing the model for an operating system and present a proof-of-concept implementation for important subsets of the FreeBSD operating system. Section 5 shows case study and performance results. We give conclusions and address future work in Sections 6 and 7.

2 Related Work

Keeping track of extra information in a computing system about events at the location where they occur is typically merely a problem of allocating storage for the information, when recording it. Recording the event or the nature of the event at its source is not a problem (e.g., a user modifying a file or a process receiving data from a specific remote location). However, events may influence other events and as processing of data potentially results in new events, the roles of the earlier events get lost because the information is no longer available to the entities observing the new events. If we want to keep track of which events influenced others, we need to examine how information flows within the system.

Information flow policies describe how information is supposed to be accessed or modified on a system [1, 11, 2, 3] These models merely formalize the system policy in regard to how information is supposed to flow. It is up to individual systems to make sure they adhere to the model. This may be a difficult task when considering the trade-off between security and usability on a system. In particular, to make a system practical, concessions have to be made in terms of confidentiality and integrity. In a practical environment subjects need to have more rights than they are allowed under the Bell-LaPadula and Biba models. Nevertheless, an observer might be interested in what actual information flow has taken place to determine if undesired accesses or modifications have occurred. Information flow analysis addresses this problem. Here, either the possible or actual information flow of programs is analyzed to either enforce information flow according to the model or to detect violations of the information flow policy.

2.1 Static Information Flow Analysis

A substantial amount of research has been done in the area of static information flow analysis [36]. The primary focus lies in assuring data confidentiality and integrity when using certain programs on a system. One approach is to use techniques from type systems for controlling information flow. Security identifiers are attached to variables and expressions and used to verify the information flow at compile time [12, 13, 43, 20, 21, 35, 37]. Other approaches use semantic-based security models [42, 24],

analyzing end-to-end program behavior, often related to some sort of noninterference [19, 32] policy.

Static information flow analysis is a powerful method to determine how information is propagated by which principals. However, all the programs running on a system need to be analyzed to verify that they adhere to the information flow policy. Even though advances are made in the automation of the tedious work, analyzing programs is still a time-consuming and expensive task that increases with a program's complexity. Furthermore, one can only be certain about those programs on a system that have been analyzed, which takes away the ability to execute general-purpose programs. This limitation is acceptable in many scenarios where knowledge of the exact information dissemination in a system is crucial.

2.2 Dynamic Analysis

Some work exists that is intended to track runtime information flow of programs [44]. Here it is proposed to incorporate Denning's compiler-based information flow concepts [12, 13] into the Java virtual machine to keep track of every user identifier associated with the running program and then enforcing access control at the time when system calls need to be made. A weakness of this approach is that the user identifiers are vulnerable to tampering when kept in user space. Also, no implementation or results have yet been published.

The Data Mark Machine developed by Fenton [16] associates a security class label with every variable on the system and is able to analyze information flow at execution time. However, it is a highly abstracted machine not suitable to monitor information flow on a more complex system.

Other work utilizes virtual machines that record certain checkpoints of the system state they are emulating, thus allowing a post-event analysis of how information has disseminated through the system. This work is either motivated by intrusion analysis [15, 18, 26] or operating system and program debugging [27, 45, 46]. This logging of state changes allows an investigator to retrace the operations and information flow as it has occured on a system. However, to determine if a specific resource has been affected by certain input or actions, a separate trace graph has to be constructed for the resource. Furthermore, to list all resources that were affected by a specific input or action, trace graphs for all resources must be constructed and the results be filtered afterward.

The information flow analysis techniques we discuss here find their application in specialized environments. To have a complete understanding of how information flows, all the programs running on the system need to be analyzed. The alternative to this is to limit the execution of programs to an environment where their actions can be monitored. Thus far, research in dynamic information flow analysis has utilized program interpreters of virtual machines for that purpose, which comes at a high

performance cost. In this paper we will present an alternative that allows us to track information across a system dynamically without the performance penalty incurred by virtual machines or interpreted programs.

3 Propagation Model

The propagation model we introduce in this section is meant to reflect the methods a (computing) system has to observe the information flow of its principals (processes). There are similarities to Denning's lattice model [11]. We do not, however, have a concept of a flow relation, which governs permissible information flow. We also do not attempt to enforce any security of the information flow on the system, but want to merely observe it. Denning talks about dynamic binding as not being practical, because the objects and processes of the model "are usually considered to have a fixed class." The approach we present in this paper utilizes labels (similar to Denning's security classes) to be dynamically bound to our principals and objects at run-time. While Denning is clearly concerned about objects belonging to particular security classes, we use our labels as tokens that are propagated during communication exchange. Furthermore, we only support a limited set of operations, because these are the only ones available to a system that passively observes information exchange of its principals. We now formally describe our model.

3.1 Causality and Labels

Consider a system that is comprised of active principals and passive objects. We define a *principal* as the active agent on a system that performs actions and interacts with other principals. A principal may act as an agent of a human being, on behalf of other principals, or the system itself. Principals can create other principals; create, access, and modify passive objects; and exchange information with other principals through communication channels. We use the term *subject* to denote either a principal or an object.

Principals have inputs and outputs for the purpose of interacting with other principals and accessing passive objects. Observable changes in a principal's state are defined by its outputs. Such a change can be caused by many factors: implicit measures within the principal itself, input from the system, input from another principal, input from an object, or any combination of the above. There might be other changes of state for a principal. These may include hardware failure, electrical surges, or cosmic radiation. As these are non-deterministic, unpredictable events, those types of change of a principal's state will be ignored by our model.

We define a given input to be a *cause* for an observable change in state of a principal (and thus the cause for an output), if a change in state observed in the output is different based on whether or not the input is provided. This is consistent with the principle of non-interference [19, 32].

When all the internals of a principal are known and deterministic, it is possible to analyze exactly which inputs cause what outputs. However, generally this is not the case. If we view a principal as a black box and only observe its inputs and outputs, even in the simple case when there are only two inputs and one output, it is undecidable to determine which input (if any) caused the output, for the general case [5].

To avoid the undecidability issue, we utilize a pessimistic heuristic: if an output can be observed for a principal at time t, we consider all previous inputs of time $t_i \leq t$ as potentially having caused the output. Thus any information exchange between principals – direct or indirect – has a potential effect on successive outputs of a principal. This approach will yield false positives as certain inputs may not have been the cause of an output. However, this ensures that any input that did cause an output will be considered.

In the following, we will present a model that allows us to track communication between principals based on causality. For this purpose we introduce an operation that binds a *label* to a principal. By label we mean an arbitrary string of bytes whose interpretation depends on the given application of the model. Labels are *propagated* to principals and objects based on causality: if an input causes an output, the label of the input's source needs to be propagated to the input's target. By propagation we mean some function of the label and any existing labels of the target (the target's *label set*), resulting in a new label set for the target.

This approach differs from the information flow analysis methods we discussed in Section 2. It is a dynamic run-time solution without the overhead incurred with utilizing specialized interpreters or virtual machines. We achieve this at the penalty of being imprecise, meaning we do not track the exact information flow but rather all possible ones. The dynamic approach gives us the advantage of being able to execute arbitrary programs on a system without having to analyze their information flow first. There is no need for an interpreter-based runtime environment or virtual machines, which incur a performance penalty. Also, no special hardware is needed to track information flow. Using labels and propagating them dynamically at run time means that we perform a forward propagation of meta-information. As a result, we can observe the information immediately with the principals and objects on a system, without having to perform any reconstruction steps as required in other work, where the operations of a system are recorded and later analyzed [26]. With our approach it is possible to not only determine labels sets of any given principals and objects, but also to determine if a given label is present at what entities without having to reconstruct the information flow explicitly for all entities on the system.

3.2 Description of the Model

The ability to bind arbitrary labels to principals allows us to address a variety of scenarios where it is important to track not only how information propagates on a

system but also what kind of information. It further allows us to focus on only the information that is relevant for a given scenario.

The following is an abbreviated description of the propagation model. In this paper we only focus on the basic model and its properties. A full description and model variants discussing space considerations have been addressed in previous work [5].

Information exchange between principals is performed through communication channels that are established between the two participants. As the communication between principals does not necessarily have to be synchronized, there may be an arbitrary time delay between one participant's *write* and the other's *read* operation. In a sense, channels are abstract passive objects that act like FIFO queues from which principals read and write. Channels in this model are uni-directional, meaning that only one principal may write to the channel and the other can only read from the channel.

Information between principals may also be exchanged indirectly through storage objects (objects). By storage objects we mean shared objects on a system that are used to store data either temporarily or on a long-term basis. As objects may be used to transfer information, labels from principals also need to be associated with storage objects that are modified or created by them, and be propagated to principals accessing those objects. Here, the concept of information exchange is less restrictive than above, as only a modification of the object or actual data transfer from the object needs to be considered. The mere existence of an object may also be used to exchange information between principals one bit at a time: one principal can create or destroy an object while a second principal tests for the object's existence. If the two principals synchronize their operations, they can exchange information this way. This is the type of channel Lampson defines as a *storage channel* [29, 30].

We define the following sets for the propagation model:

L**:** set of labels

P**:** set of principals

O**:** set of storage objects

$C \subseteq \{P \times \{P \cup O\}\} \cup \{\{P \cup O\} \times P\}$**:** set of ordered pairs $\langle i, j \rangle \in C$ if and only if a communication channels exists between $i \in P$ and $j \in \{P \cup O\}$ or $i \in \{P \cup O\}$ and $j \in P$.

label()**:** $P \cup O \rightarrow 2^L$, a function that given a principal or an object returns a subset of L that is called the *label set* of the principal or object. The function label(ϕ) will always return the empty set $\emptyset$.

clabel()**:** $O \rightarrow 2^L$, a function that given an object returns a subset of L that is called the *creator label set* of the object. This function denotes the label set of the object's creating principal at the time of creation.

Channels between two principals are uni-directional and function according to the consumer-producer model as a FIFO queue. Instead of actual data items, it is sufficient for this model to require only that the label set associated with the data is contained in the channel. For this purpose, we define two operations on channels:

enqueue(c, l) adds a label set $l \in 2^L$ to the FIFO queue of channel $c \in C$.

dequeue(c) returns and removes the next label set $l \in 2^L$ from the FIFO queue for $c \in C$. If the queue is empty, the empty set is returned.

We further have a mapping on label sets, **update**: $2^L \times 2^L \rightarrow 2^L$. The mapping determines how label sets are updated as two processes exchange information. This function must be defined for sub-models derived from this model.

Below we define the operations on the sets described above. An operation consists of two parts: an optional precondition and an action part. If there is a precondition associated with an operation, it must be fulfilled for the operation to succeed. Otherwise, the operation fails. Operations without a precondition will always succeed. Only a principal may perform an operation. To avoid a cumbersome notation, the principal that performs the operation can either be implied from the operation itself, or, if necessary, is explicitly mentioned.

create(p_1, p_2) Principal p_1 creates principal p_2. The label set of p_1 needs to be inherited by p_2:

$$\begin{aligned} P &:= P \cup \{p_2\} \\ \text{label}(p_2) &:= \text{label}(p_1) \end{aligned}$$

create(p, o) Principal p creates object o. The object's label set as well as the creator's label set need to be inherited from p:

$$\begin{aligned} O &:= O \cup \{o\} \\ \text{clabel}(o) &:= \text{label}(o) := \text{label}(p) \end{aligned}$$

open(p_1, p_2) The channel $\langle p_1, p_2 \rangle$ is opened between principals p_1 and p_2. The channel has the direction from p_1 to p_2, meaning that p_1 can perform write operations and p_2 can perform read operations on the channel. Whether the operation of opening the channel succeeded or not can already be viewed as the exchange of 1 bit of information: success or failure. Therefore, the label sets of both principals need to be updated at this point:

$$\begin{aligned} C &:= C \cup \{\langle p_1, p_2 \rangle\} \\ \text{label}(p_1) &:= \text{label}(p_2) := \text{update}(\text{label}(p_1), \text{label}(p_2)) \end{aligned}$$

open(p, o) The channel $\langle p, o\rangle$ is opened between principal p and object o. P can write to the object. A successful open indicates that o actually exists, so the object's creator label set needs to be updated with the principal's:

$$\begin{aligned} C &:= C \cup \{\langle p, o\rangle\} \\ \text{label}(p) &:= \text{update}(\text{label}(p), \text{clabel}(o)) \end{aligned}$$

open(o, p) This is analogous to the previous operation, using channel $\langle o, p\rangle$ and p having read access instead.

write(p_1, p_2, n) Principal p_1 writes n data items to the channel $\langle p_1, p_2\rangle$, i.e. it consists of n enqueue operations. In this case the channel $\langle p_1, p_2\rangle$ needs to be open.

Precondition: $\langle p_1, p_2\rangle \in C$

repeat n times:

$$\text{enqueue}(\langle p_1, p_2\rangle, \text{label}(p_1))$$

write(p, o) Principal p writes data to object o. In this case the channel $\langle p, o\rangle$ needs to be open. Because o is receiving information, o's label set needs to be updated with p's:

Precondition: $\langle p, o\rangle \in C$

$$\text{label}(o) := \text{update}(\text{label}(p), \text{label}(o))$$

read(p_2, p_1, n) Principal p_1 reads and removes n data items from channel $\langle p_2, p_1\rangle$, i.e. it performs n successive dequeue operations.. In this case the channel $\langle p_2, p_1\rangle$ needs to be open. Because p_1 is receiving information, p_1's label set needs to be updated with the ones read from the channel:

Precondition: $\langle p_2, p_1\rangle \in C$

repeat n times:

$$\text{label}(p_1) := \text{update}(\text{label}(p_1), \text{dequeue}(\langle p_2, p_1\rangle))$$

read(o, p) Principal p reads data from object o. In this case the channel $\langle o, p\rangle$ needs to be open. Because p is receiving information, p's label set needs to be updated with o's:

Precondition: $\langle o, p\rangle \in C$

$$\text{label}(p) := \text{update}(\text{label}(p), \text{label}(o))$$

close(p_1, p_2) The channel $\langle p_1, p_2 \rangle$ between principals p_1 and p_2 is closed. Both principals may interpret this event, so this can be seen as a 1-bit information exchange. Both principals' label sets need to be updated with each other's label sets:

Precondition: $\langle p_1, p_2 \rangle \in C$

$$\begin{aligned} C &:= C - \langle p_1, p_2 \rangle \\ \text{label}(p_1) &:= \text{label}(p_2) := \text{update}(\text{label}(p_1), \text{label}(p_2)) \end{aligned}$$

close(p, o) The channel $\langle p, o \rangle$ between principal p and object o is closed. No information is exchanged.

Precondition: $\langle p, o \rangle \in C$

$$C := C - \langle p, o \rangle$$

close(o, p) This is analogous to the previous operation, using channel $\langle o, p \rangle$ instead.

addlabel(p, l) Label l is bound to principal p. L needs to be added to any existing labels in p's label set:

$$\text{label}(p) := \text{label}(p) \cup \{l\}$$

destroy(p) Principal p is destroyed. All open channels involving p are closed.

$$\begin{aligned} &P := P - p \\ &\{\langle x, y \rangle \in C | x = p \vee y = p\} : \text{close}(x, y) \end{aligned}$$

destroy(o) Object o is destroyed. All open channels involving o are closed.

$$\begin{aligned} &O := O - o \\ &\{\langle x, y \rangle \in C | x = o \vee y = o\} : \text{close}(x, y) \end{aligned}$$

A sequence of operations is an ordered list of operations, in the order they occur. At any given discrete time interval t_i exactly one operation is allowed and that operation is considered atomic. To extend the model, it may be necessary to define composite operations from the basic operations given above. For example, the opening of a bi-directional channel between principals p_1 and p_2 may be defined as $\{\text{open}(p_1, p_2), \text{open}(p_2, p_1)\}$. This new operation is also atomic. There is also an initial state of the system, which at a minimum consists of $P = \{p_0\}$ and $\text{label}(p_0) = \{\}$.

3.3 Properties of the Propagation Model

For the remainder of this paper, we require that the label-updating function *update* preserves the labels. An example of this is the set union operation, i.e., update = $\cup$. Furthermore, we define a special set $P_g \subset P$ of principals, which contains the principals that may generate labels.

Such a group P_g of principals implies that each label is uniquely identifiable as having originated from a particular principal or a group of principals from P_g. That means there is a mapping $originated : L \rightarrow 2^{P_g}$ that takes a label and returns the principal that created the label or the group of principals from which the label could have originated. For example, if we have a group of principals `httpd, telnetd, ftpd, sshd`, which are responsible for communicating with principals from other systems and they generate labels identifying those systems, then any label found within the system that is an identifier of a foreign system must have been generated by one of those four principals.

We shall now give an informal description of what we consider information exchange between principals. Intuitively, two principals exchange information at or after a given time when data flows from one to the other. If intermediaries, such as other principals or objects, have been used to exchange the information, then information is exchanged between the source and the intermediaries as well as the intermediaries and the target. The successful opening of channels constitutes a 1 bit information exchange. In case of checking the existence of an object for the purpose of information exchange is only relevant for the creator of the object and the principal who tests the existence. Principals who may have written to the object in the meantime do not matter as they played no role in the existence of the object. Writing data into a channel is only relevant if that very data is also read. That means that if data is already present in the channel prior to a write operation w, that data needs to be read from the channel before a reading principal is affected by w.

Because of our conservative approach when tracking information flow, we can not be sure that information was actually exchanged between principals. However, information could have been exchanged. Therefore, we consider a *potential information exchange path* $IE_n(p_1, p_2)$ between two principals $p_1, p_2 \in P$ as a sequence of $n \geq 1$ operations from our model

$$\mathrm{op}_1(p_1, s_1) \circ \mathrm{op}_2(s_1, s_2) \circ \ldots \circ \mathrm{op}_n(s_{n-1}, p_2)$$

that directly adheres to the above description. For a more formal description we refer to previous work [5].

Some operations may only be in the path if they are matched with other operations in a manner so that information is propagated from one subject to the next. A write operation needs to be matched with sufficient read operations, an open operation of a channel between principals is only relevant for the information exchange if the opening principal was created by an operation in the path, and an open operation of a

channel involving an object is only relevant if the object was created by an operation in the path. In general these operations may exist by themselves, but without their matching operations will not contribute to the information exchange and therefore need not be in the path.

We say p_1 *and* p_2 *potentially exchange information* iff there exists a potential information exchange path between p_1 and p_2.

Given the definitions above, our model has the following two properties[1]:

1. If information is exchanged between principals $p_1 \in P_g$ and $p_2 \notin P_g$, and label $l \in \text{label}(p_1)$ prior to the information exchange, then $l \in \text{label}(p_2)$ after the information exchange.

2. If principal $p_2 \notin P_g$ and label $l \in \text{label}(p_2)$, then information was potentially exchanged between p_2 and a principal $p_1 \in \text{originated}(l)$.

4 Implementation

In the following we discuss the implementation of our model as a proof-of-concept study. The model was implemented by modifying the FreeBSD 4.12 operating system [17]. The implementation is kernel-based, which means that the propagation method lies in a protected space that cannot be tampered with from user mode programs. If the kernel can be trusted, then so can the label propagation and any information gained by it. Implementing the model for a real production operating system as opposed to a simulated one serves multiple purposes:

- We demonstrate that utilizing the model is feasible for modern operating systems.
- We can accurately measure the computational overhead needed for label propagation to work.
- We encounter and can address the difficulties and limitations that come with such an implementation.
- Results obtained from this proof-of-concept implementation may be applied to other operating systems with similar architectures.

Section 4.1 describes which subsystems would need to be addressed for a full implementation of the model we described. Given that our implementation is merely a proof-of-concept, we focus on the major subsystems needed for label propagation to function. It addresses all the important aspects that need to be considered, and

[1] A proof of the properties can also be found in previous work [5]

most of the other subsystems can be implemented in a similar fashion. In the subsequent sections we describe the parts that were actually implemented, namely: the data structures and operations introduced to the kernel (Section 4.2), how to modify network sockets as part of interprocess communication (Section 4.3), and how to handle label propagation for files (Section 4.4).

4.1 Subsystems Affected by Label Propagation

Applying our model for a computing system means that we need to translate the sets of principals, objects, and channels to entities of the system. Furthermore, the operations of the model need to be implemented accordingly. The acting principals on a computing system are its processes. Therefore, the set P maps to the set of processes on the system. This is consistent with the notion that on all computing systems, there is one initial process (such as `init` in the UNIX world), from which subsequently all other processes on the system are created (unless it is not a multi-process system, in which case there is only one initial process). The set of objects O of the model are all the resources that are shared on the system among processes. Those resources that belong solely to one given process need not be considered. Furthermore, the set of possible channels is comprised of the system's provisions for interprocess communication, as well as the means to access, modify, and create shared resources. To implement label propagation for an entire operating system, essentially three parts need to be completed:

1. The system needs to be aware of labels and be able to bind them to its principals.
2. All interprocess communication needs to be covered by the label propagation mechanism.
3. All objects shared among processes need to be associated with labels for the duration of the sharing and those labels updated according to the model.

4.1.1 Shared Resources

The resources that are shared among processes on a system are numerous. In FreeBSD and other UNIX-type operating systems they consist of at least the following[39, 5]:

- Files
- Mutexes and Condition Variables
- Read-write and Record Locks
- Semaphores
- Shared Memory

- Sockets and Message Queues

The implementation we describe in the following handles label propagation for a fixed number of globally shared labels. That means that there is an upper bound on the number of labels, which allows us to allocate sufficient space for the processes and objects or fail during their creation. Furthermore, the update function will always succeed and we do not have to worry about the measures to take when there is insufficient space for labels as a result of the update.

As there is no system involvement when shared memory is processed, there is no elegant solution to keep track of label propagation. One might imagine monitoring all the system calls that involve copying of data between buffers and determine if any of the addresses involved fall into a shared memory region. However, because data may directly be assigned in chunks of up to a word length, this will not cover all of the information flow between processes. Barring the utilization of static analysis techniques as discussed in Section 2.1, a monitoring device such as Fenton's Memory Mark machine [16], or the utilization of virtual machines or special hardware, the only reliable method to properly implement the model is once again conservative: all processes that share memory between them need to be treated as a single principal during the duration of the sharing. For an implementation of this, for every operation that involves a principal, further lookups will have to be made to see if it belongs to such a group and then perform the operation for all processes involved. The utilization of shared memory is a feature not common to many programs. Thus, the inclusion of such a mechanism lies well outside of a proof-of-concept implementation, and we will not consider shared memory any further (and as a direct consequence we also will not have to consider mutexes, condition variables, and read-write locks)[5].

4.1.2 Interprocess Communication

Interprocess communication mechanisms allow two processes on a system to exchange data. In UNIX-like operating systems there are the following mechanisms available [31, 39]:

- Pipes and FIFOs
- Sockets
- Message Queues
- Remote Procedure Calls
- Signals

4.1.3 Operations and System Calls

In FreeBSD and many other operating systems the way a process is able to communicate with other processes or access system resources is governed through the use of system calls. System calls are the interface from user space processes and the operating system kernel. The shared objects are created and manipulated that way and all the interprocess communication mechanisms utilize system calls. Due to space constraints we are unable to discuss at large the details of our implementation. For a more detailed description we refer to previous work [5, 9]. In the following we give a rough outline of the implementation efforts.

Table 1: FreeBSD system calls and their relevance to the model

Operation	System calls
Create	`fork, open, creat, mknode, fcntl, sem_open, semget, sem_init`
Open	`pipe, socket, socketpair, accept, msget, open`
Write	`write, writev, send, sendto, sendmsg, fnctl, ioctl, setsockopt, msgsnd, msgctl, kill, sem_wait, sem_trywait, sem_post, semget, semop, semctl, pwrite, open, ioctl, truncate, ftruncate, chmod, fchmod, lchmod, chflags, fchflags, chown, fchown, lchown, utimes, lutimes, futimes, rename, link, symlink, mkdir`
Read	`read, readv, recv, recvfrom, recvmsg, fcntl, iocntl, msgget, msgrcv, msgctl, sem_trywait, sem_getvalue, semget, semop, semctl, pread, stat, lstat, fstat, poll, access, chdir, fchdir, readlink`
Close and destroy	`close, pclose, shutdown, msgctl, kill`

Table 1 gives an overview of which FreeBSD system calls are relevant for which operation of our model. The number of system calls we have listed there that need to be modified to allow full label propagation is extensive. Some of the implementation work may be reduced as some system calls share lower level functions in the kernel, where the appropriate modifications may be performed (see the following sections). However, most of the complexity arises from the need to address communication through storage channels. In the general case processes will not abuse mechanisms such as the existence of files, locks, semaphores, or message queues, changes in socket parameters, or changes in file metadata to exchange information. With the introduction of observation techniques such as the ones we discuss in this paper, this may

change, however. If all legitimate channels on a system are effectively monitored then malicious users will find ways to circumvent those channels. However, addressing those storage channels in a proof-of-concept implementation is well beyond the scope of the work we present here. Therefore, we picked the most relevant subsystems and focused only on implementing label propagation for the data channels on the system. That means that we will ignore the open and close operations of our model, and do not keep track of creator label sets. The techniques we describe in the following should apply for most of those instances, and many of the system calls we do not address can be modified in the same manner as we describe below.

4.2 Data Structures and Operations

The main data structure that manages the labels in the kernel is a global table, which is an array that contains the label data as well as its position within the table. This allows the referencing of a label to be its entry number within the label table. Thus, any given label set associated with a process or object can be as simple as a bit-vector, where each bit signifies whether the label at that position is contained in the set or not. That means that any given label set is a bit vector with as many bits as there are possible labels. Note that the actual content of the label is not important for the propagation at all. Any interpretation of the labels is done by other extensions of the system (i.e., access control mechanisms, logging facilities, etc.), which are outside the scope of this work.

It is conceivable to have a further mapping that, given a label, finds the position in the table. This may be useful to determine if a potentially new label to the system is already contained in the label table (and thus is not new). This may be achieved, for example, with a binary search tree using the label data as a key that maps back to the position in the global table. The global table will still be necessary, because once a position has been assigned to a label it must be permanent or the label sets will carry incorrect information. Plus, we achieve a $O(1)$ lookup for a given position in the table as opposed to the $O(\log n)$ such an operation would take if only a search tree were realized.

To provide an interface to user space that allows user programs to query processes' labels, two new system calls were introduced to the system. The `getlabel` call retrieves the label set for a specified process id and stores it into the supplied buffer. The `getlabeldata` call retrieves the label data for the specified entry in the global label table and stores it in the supplied buffer. It is thus the responsibility of the user space program to compute the position in the label table from the bit vector and then retrieve the data.

For debugging purposes we also implemented another system call, `addlabel`, which binds a new label to a specified process. This call would not be part of a regular system.

Note that there are no operations removing labels from the system. That means

that the proof-of-concept implementation will only be useful to a limited degree for label sets that are not fixed in size. This could be remedied by introducing a reference count for each label and, once the reference count reaches zero, removing the label from the table. However, this would imply that the update function no longer is a simple "OR" operation. The `update` function now would need to determine which labels were newly added to label sets as part of the update and increase the reference count. Furthermore, whenever a process, shared resource, or label set residing in a channel gets destroyed the reference count needs to be decreased.

4.3 IPC: Sockets

From the different types of interprocess communication described in Section 4.1.2, the socket subsystem is the most complex one. The FreeBSD implementation of pipes utilizes supposedly the socket infrastructure to transmit data [31], but our performance overhead results in Section 5.4 indicate otherwise. Furthermore, the Sun RPC mechanism also uses sockets and the network subsystem to function. That and the fact that message queues are commonly not used too often by programs led us to implement label propagation for the socket subsystem only.

The system calls that we need to consider for an implementation are `read`, `readv`, `recv`, `recvfrom`, `recvmsg`, `write`, `writev`, `send`, `sendto`, and `sendmsg`. However, there are lower level functions that are invoked by those system calls, which, in turn, all call the `soreceive` function for the reading calls and `sosend` for the writing calls. Figure 1 illustrates the function hierarchy and the overall network stack structure of FreeBSD.

We modified the `sosend` and `soreceive` functions to prepare the mbufs as described above with some minor changes to the `soo_read` and `recvit` functions, as well.

4.4 Shared Resources: Files

Files are by far the most complex among the shared resources in FreeBSD. Furthermore, files are the only kind of shared object in the system that are intended for data exchange (we do exclude shared memory from our analysis; see Section 4.1.1). Thus our proof-of-concept implementation needs to deal only with files for label propagation among shared resources.

FreeBSD, like many other UNIX-like operating systems, supports many different file systems. To bring many different file systems with different layouts and operation into one single framework transparent to the user space processes, FreeBSD utilizes a virtual file system layer. This ensures that standard operations that the operating system supplies can be properly mapped to the file system specific functions. The generic operations that can be performed on a file descriptor by a process are mapped in a structure, which contains pointers to the appropriate low-level operation for

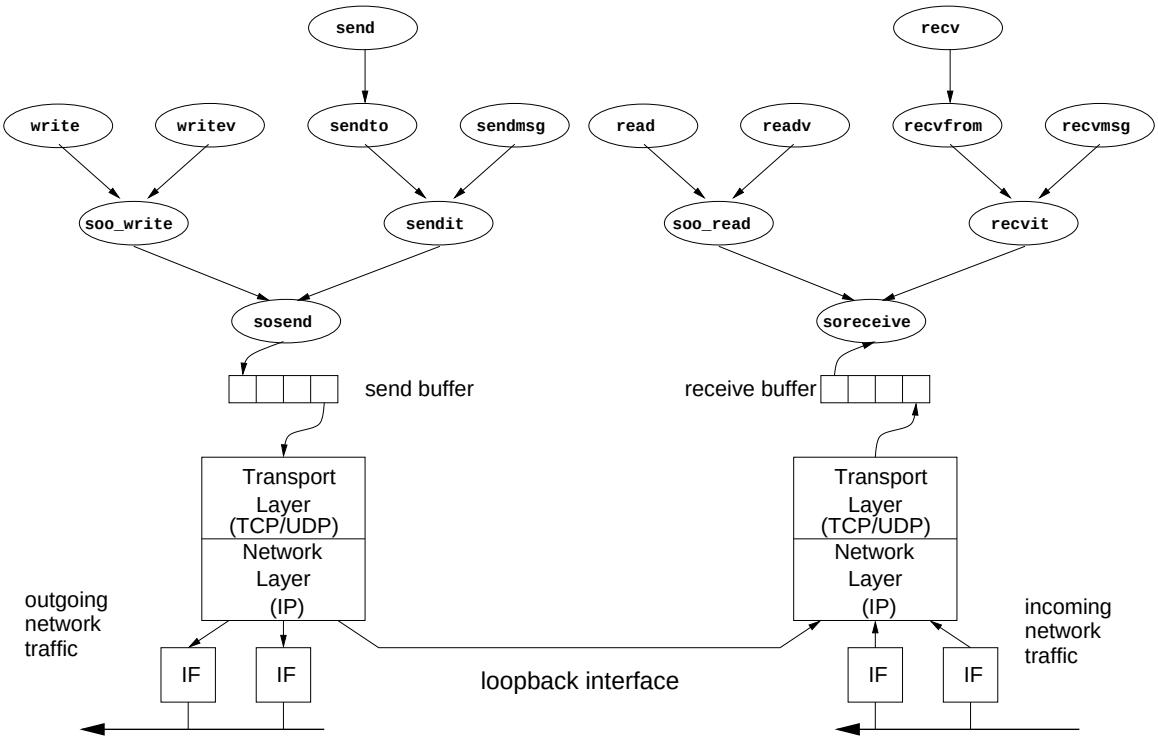

Figure 1: FreeBSD kernel functions for socket I/O

the underlying file system. This is done for read, write, ioctl, poll, stat, and close operations.

There are two ways to implement label propagation regarding file systems: the labels are stored with each file, or the kernel itself keeps track of which file is associated with what labels. The first approach has the advantage that there is little computational overhead as there needs to be no explicit mapping of files to labels. Simply by accessing the file, which is done anyway during the operations, the label set may be retrieved. Plus, the labels for a file are automatically stored permanently. However, this way each individual file system needs to be modified to accommodate label propagation. This might not be possible for certain file systems. While for some file systems, such as *ext2* [10], there are unused fields (e.g., the access control pointer), which can be used to point to blocks containing labels, there is no such extra space in the Reiser file system [4], and future file systems might be similarly frugal with the space they use. When the kernel keeps track of what the labels for a file are, then there needs to be some sort of lookup data structure for the mapping. However, by keeping the mapping in the kernel, the implementation is independent of the underlying file systems, supporting any of those supported by the operating system itself, provided a unique identifier can be assigned to each file on the system. Furthermore, a data structure that contains all the files that have label sets associated with them

allows us to efficiently answer questions such as: "Which are the files that possess label X?". If labels were stored directly with the file, then all the files on the system would have to be examined, not only those that have a label set.

Because of those advantages and the fact that an implementation affects fewer subsystems of the kernel, we have decided to keep track of file label sets directly in the kernel. For this, we have implemented a global mapping that utilizes a balanced binary search tree. This gives us $O(\log n)$ lookup and insertion time, where n is the number of files with labels.

Files in FreeBSD (and according to the POSIX standard [22]) are uniquely identified system-wide by the pair of device and inode numbers. This is because, inode numbers are unique only within the disk partition where they reside. Both are currently 32-bit values, so there is a hard upper limit of 64 comparisons for up to 2^{64} total items when performing operations on the binary search tree.

The system calls that are responsible for data transfer to and from files are `read`, `readv`, `write`, `writev`. The `read` and `readv` system calls, both invoke `fo_read`, which invokes the proper low-level function for read (this is true not only for regular files, but also for sockets and pipes). Therefore this is the place to handle the label propagation. After a successful call to the low-level read function we retrieve the device and inode information. If an entry for the device/inode pair exists, then we update the calling process's label set with the one retrieved by the lookup.

Similar to the reading calls, the `fo_write` function is called by the writing system calls. After a successful low-level write operation and if the calling process has a label set associated with it, we perform a lookup and update the retrieved label set with that of the writing process.

5 Results

In the following we demonstrate the effectiveness of our approach by showing how to use the implementation to solve some of the problems discussed earlier. Furthermore, we will measure performance overhead to show that an implementation of label propagation is feasible.

5.1 User Influence

We shall demonstrate the effectiveness of user influence labels by using the following example. As depicted in Figure 2, the process controlled by User A reads data from File 1. It then communicates via interprocess communication (IPC) with a process controlled by User B, which subsequently creates a new file with the content of what was communicated during the exchange with User A.

To associate a user identifier with the processes that are under that user's control, we bind the user ID that is assigned to the process at login time to the process as a label as well. While this may seem redundant at first, note that the user ID of a

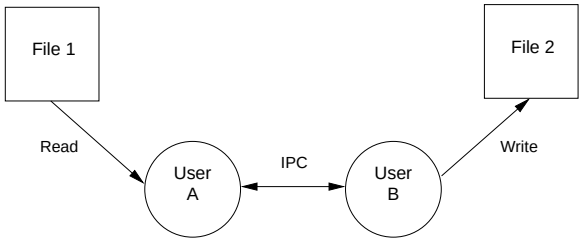

Figure 2: The contents of File 2 are influenced by User A

process that is recorded in the process table is subject to change during subsequent login and logout operations (e.g., via the `su` command), while a label is persistent. The `setlogin` system call was modified to add the user identifier as a label to the process's label set that invoked the `setlogin` call.

To illustrate that any labels already attached to the original file will also be propagated, we first execute the `set_filelabel` system call to bind a label to the file. Then we execute a program that opens the file, reads data from it and then opens a TCP connection to a second process with a different user id and transmits the file. The program also prints out the label set to the console before and after each operation. The second process (listening on the TCP socket) reads the data from the first process and then creates a file with the data it received. Finally, we execute a program that calls `get_filelabel` to see what the label set of the newly created file is. The output from both user sessions are shown in Figure 3.

The `setfilelabel` program binds the label "File XXX" to the file XXX on the system. The `sender` program opens the file specified on the command line, reads a number of bytes from it, then opens a TCP connection to port 7000 on the local host and transmits the data that was read from the file. The `receiver` program listens for TCP connections on port 7000. Once a connection is established, it reads a number of bytes from the socket and then creates a new file with the data it received. The `sender` and `receiver` program also print out their label sets before and after the operations that receive data.

Initially the two processes have only the user id label bound to them.[2] After reading the file, Process A now also contains the label "File XXX". After receiving data from the network socket from Process A, Process B has three labels associated with it: "User 1002", its original label, as well as labels "User 1001" and "File XXX". After Process B creates and writes data to file YYY, the `filelabels` program reveals that the label set of file YYY also contains those three labels. An investigator examining file YYY now can determine that both users 1001 and 1002 could have played a role in the current state of the file, plus that there is a possibility that the contents of file XXX might also have been an influence. We can further conclude that

[2]For better readability we actually use a string 'User nnnn' as a label as opposed to a 2-byte label containing only the id itself.

```
          SESSION A                                   SESSION B

% ./setfilelabel 265476 649649 "File XXX"
                                                  % ./receiver YYY
                                                  Receiver starting with pid 162
                                                  Label set at start:
                                                  Label 0: User 1002
                                                  Listening on port 7000
% ./sender XXX
Sender process starting with pid 163
Label set:
Label 1: User 1001
Label set after reading file:
Label 1: User 1001
Label 2: File XXX
Writing data to socket
                                                  Connection established
                                                  Label set after socket read:
                                                  Label 0: User 1002
                                                  Label 1: User 1001
                                                  Label 2: File XXX
                                                  Writing to file YYY

                                                  % ./filelabels YYY
                                                  File YYY has labels:
                                                  Label 0: User 1002
                                                  Label 1: User 1001
                                                  Label 2: File XXX
```

Figure 3: Output from the user influence test from both user sessions

no other users could have been responsible in the creation or modification of the file YYY.

5.2 Location Information

To show that the label propagation also works well with non-custom programs, we also tested the location information case study as described in previous work [8]. Here, we assign origin information to a file containing C-source code, a library used in the code, the `gcc` compiler, and the current session as shown in Figure 4.

We do this manually through utility programs that call the system calls `set_filelabels` and `addlabel` and then compile the program. Then we use the program from the previous example to print out the label set of the newly created file (see Figure 5).

As in the example, we bind a label called "OS-CDRom" to "/usr/bin/gcc", "Console" to the "sender.c" file, "Website" to the file "printutils.c", which we compile in directly as opposed to implementing a library from it first; the result is the same. Plus, we bind a label "192.168.0.1" to the shell process. We then compile the "sender.c" program. A `filelabels` lookup on the file "sender" now shows that the file is associated with all of the labels of the entities that played a role in its creation.

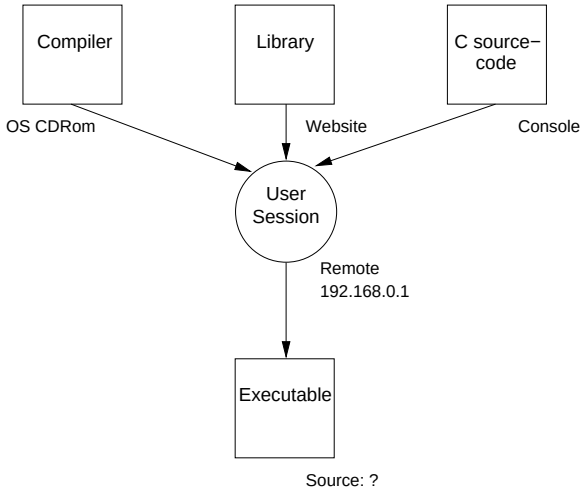

Figure 4: Given the origin of the involved entities, what is the origin of the new file?

5.3 Remote System Compromise

A system is *compromised*, when an unauthorized user has gained control over the system. This is typically done by exploiting a vulnerability of the system to receive access to the system and permissions to perform certain tasks. In many cases, the compromise occurs through the exploit of a vulnerability of one of the system's network services. Our label propagation mechanism does not differentiate between authorized and unauthorized accesses and operations. If we are able to bind a location label to the process accepting the network traffic, then those labels are propagated for both legitimate as well as malevolent uses. If a real compromise occurs from a remote location, all processes and files affected by it will be labeled with that location label.

To simulate a system compromise we will run a program that accepts network connections and supplies a shell. This is the basic functionality of a backdoor program, but could also be the result of the compromise of a well-known server daemon process (be it `httpd` running as the http user, or even `sshd` running as the root user). The nature of the compromise (e.g. buffer overflow or script vulnerabilities) is not important, the end result is the same: a remote attacker has access to a shell with the privileges of the daemon process that was compromised. Thus it is sufficient to test the label propagation with a "normal" remote login via `ssh`.

For our labeling approach to capture the entry point of the intrusion, we need to associate network-location labels to those processes receiving data from the network. For this, we have modified the `accept` system call to bind a label to the process invoking `accept` whenever a connection is successfully accepted. For demonstration purposes, we only use the foreign IP address as a label. For a more complete network identifier, the 4-tuple of foreign IP address and port as well as the protocol and local

```
% ./setfilelabel sender.c Console
% ./setfilelabel /usr/bin/gcc OS-CDRom
% ./setfilelabel printutils.c Website
% ./setproclabel -1 192.168.0.1

% gcc -o sender sender.c printutils.c

% ./filelabels sender
File sender has labels:
Label 0: Console
Label 1: OS-CDRom
Label 2: Website
Label 3: 192.168.0.1
```

Figure 5: Output from the location information case study

port information can be used. For UDP, a similar addition can be made to the `recv` and `recvfrom` system calls.

Figure 6 shows an `ssh` session with location labels enabled. The process running the `proclabels` program is clearly marked with the location label[3], which means that the process running the shell also carries the label. Furthermore, if we create new files on the system or modify existing ones, the label is propagated.

```
florian@schlaraffenland:~> ssh morpheus-8
Password:
Last login: Tue May 17 14:09:40 2005 from schlaraffenland
Welcome to FreeBSD!
%./proclabels
Process ID: 74778
Label 0: 128.10.243.68
%echo test > testfile
%./filelabel testfile
File testfile has device 0x40d04 and inode 651811
Syscall result: 0. Ret: 1
File testfile has the following labels:
Global pos: 0 data: 128.10.243.68
%
```

Figure 6: An `ssh` session with location labels

The approach we chose of binding the label at the time of `accept` has the disadvantage that the common network server architecture as described by Stevens [38]

[3]Again, we bind a string containing the IP address as a label for the purpose of better readability

has the server accept a connection and then fork a child process. The child process inherits all the connections from its parents and does the actual handling of that particular session, while the parent process goes back to the listening state. This means that over time the server process will accumulate all the labels of the past network connections and pass those labels on to its children. This can be avoided if we supply a new system call that accepts the connection, automatically forks a child process, and only then binds the label to the child. An alternative location to bind the network-location label is the interface on which the data is received. If the label is generated there and then associated with the mbufs that make up the network packet, then our socket implementation as described in Section 4.3 will automatically update the labels to exactly the processes that receive the data. The disadvantage of this is that every time data enters the system from the network we need to make a lookup whether the network-location label is already in the system or not, which may slow down performance.

5.4 Performance Overhead

In this section we will describe the results of performance overhead measurements we performed for the proof-of-concept implementation. We ran our experiments on a Sun SunFire V60x with an Intel 2.8 GHz Xeon processor, 512MB RAM, and a 36GB SCSI hard drive. As a baseline we use the generic FreeBSD 4.12 kernel that comes with the regular installation of the operating system. We then ran the performance tests on the same hardware booting the modified version of the kernel.

To measure the overall performance, we utilize the LMBench [33] benchmark suite. LMBench is a set of small micro-benchmarks, which measure system latency and bandwidth of data movement among the processor and memory, network, file system, and disk. LMBench is a widely used benchmark suite used to profile many hardware and software systems, providing more accurate results compared to other benchmarks in many cases [34]. We have broken down the tests into four categories:

1. Processor and process tests. These are tests that measure the time it takes a process to perform certain tasks. These include a basic system call (null call), the installing of a signal handler (sig inst.), the signal handler overhead (sig hand.), the time to fork a new process (fork), the time to execute a simple program (exec), and the time to execute the '/bin/sh' program (sh). Of particular interest to us are the times for the fork and the executions as these are directly affected by our modifications. The fork mechanism takes care of the label propagation by inheritance and the execute test further makes some read operations to access the specified programs.

2. File system tests. The file system tests consist of several simple tests to measure the execution time of the read system call for a file (read), the write system call (write), performing a stat operation on a file (stat), performing an fstat

operation (fstat), the opening and closing of a null file (open/close), as well as the select operation on 500 file descriptors (select). These tests use the file system cache, not the actual time it takes for the disk operation as there are too many unknown factors to consider to generate reproducible results for those. Of particular interest are the measurements of the read and write operation as they are directly affected by our modifications.

3. Network latency tests. These tests determine the time it takes for network messages to propagate. Short control messages are sent back and forth between processes and the round trip time is measured. This is done for pipes as well as sockets of types AF_UNIX, TCP, and UDP, all within the local host. All of these tests are relevant as they all measure the performance of the socket subsystem that we modified.

4. I/O bandwidth tests. The bandwidth test measure the data throughput on the system. All of these tests write a certain amount of data to a file or a communication channel in transfers of constant size. The file write test writes 8MB of data in 64K buffers. For the IPC bandwidth tests two processes are created that transfer the data between them. Pipes transfer 50M in 64K chunks, and the TCP and AF_UNIX sockets transfer 50M in 1M chunks. All of these tests are relevant as they directly measure the impact of our modifications to the I/O throughput of the system.

These tests were run on the regular FreeBSD 4.12 kernel (`FreeBSD`) as well as on three variants of our modified kernel. The first version (`Label-0`) is the kernel with all the modifications for label propagation in place but without any labels actually present in the system. This captures the overhead of a system with label propagation in place without labels present, but also gives a measure for how the socket subsystem is affected if the processes involved have no label sets associated with them. The second version (`Label-s`), in addition to the `Label-0` version also has a label associated with the shell process that invokes the LMBench test suite. It is sufficient to only bind one label to the process as the entire label vector is propagated once a process is marked to have labels. Furthermore, a small set of files is marked with labels as well. This number is initially one, but as files are created during the performance test, the number will increase slightly. This version will give a measure for the socket subsystem overhead as well as the file system performance for a small set of files with labels. The last version (`Label-l`), in addition to the `Label-s` version, has a large number of files in the system. This will give a measure for the file system overhead for a large number of files.

To label the files for the `Label-l` set, we used a program that utilizes the `set_filelabel` system call in a `for`-loop for different inode numbers to generate a set of 100,000 files – 50,000 for each of the two disk partitions on the system – that have labels associated with them. The maximum number of labels in the system was set to 1024, which

means that the label vectors are 128 bytes long. This allows us to use the label vector for user influence as the normal number of users on a system is well below 1024. We chose the maximum number of labels significantly higher than the expected number of users on a system, so we can also gain a measure for the performance overhead of a large fixed-label set implementation. Thus, our results for the user influence are pessimistic while our implementation still enables us to maintain location information for a limited number of locations.

Each test was performed 200 times for each kernel version, and the results are shown in Tables 2 through 5. Detailed measurement results including mean, standard deviation, minimum, and maximum for each version are available from previous work [5].

Table 2: Processor and process tests – times in μs

Kernel	Null call	sig inst.	sig hand.	fork	exec	sh
FreeBSD	0.4504	0.6643	1.3467	129.0783	594.1023	1176.8880
Label-0	0.4499	0.6666	1.3404	128.3902	596.1760	1173.6708
Label-s	0.4498	0.6650	1.3364	129.1161	602.4783	1185.3990
Label-l	0.4501	0.6658	1.3393	129.1359	604.5881	1187.9500

Table 2 shows that our experiments show only a slight increase in execution time for the process tests. System calls and signal handling is not affected by our modifications. The `fork` system call has to copy an extra 132 bytes, 128 for the label vector and 4 for the label flag. The measured overhead for this is less than 1%. The times for the `exec` and `/bin/sh` tests are influenced by the file system performance. The observed overhead here lies between 0.3% when no labels are present to 1.7% for the large set of labeled files (both for the `exec` test).

Table 3: File system tests – times in μs

Kernel	read	write	stat	fstat	open/close	select
FreeBSD	1.0437	0.9836	2.2547	0.6598	3.3546	19.8101
Label-0	1.1235	0.9839	2.2744	0.6649	3.4365	20.0437
Label-s	1.2139	1.3903	2.2739	0.6662	3.4457	19.9333
Label-l	1.3297	1.4934	2.3362	0.6621	3.4701	19.9283

Table 3 shows the execution times for the file system calls. As expected, there is no noticeable difference between the `FreeBSD` and the `Label-0` kernels. We measured a an average of 0.0798 μs (7.6% overhead) for the `read` test, which can be attributed to the lookup that is performed for the file. The `stat`, `fstat`, `open/close`, and `select` test did not have any noticeable differences between the kernel versions, as expected. With labels present on the system the `read` test now showed a difference of 0.1702

μs (16.3%) for the small label set and a difference of 0.2860 μs (27.4%) for the large label set. When comparing the `Label-s` and `Label-l` versions, the overhead for the 100,000 labeled files (a binary search tree depth of about 16) is 0.1831 μs (15.1%). For the `write` test, there is no noticeable difference between the `FreeBSD` and the `Label-0` versions. This was to be expected as no lookups are performed and no labels need to be inserted into the binary tree. For the remaining two versions, the `write` test was measured with an overhead of 0.4067 μs (41.3%) for `Label-s` and 0.5098 μs (51.9%) for `Label-l`, respectively. This is because first a lookup is performed to see if a label set is already present in the search tree, and if not one needs to be allocated and inserted. When comparing the small and the large label set versions, we observed an overhead of 0.1031 μs (7.4%). Note that all the observed differences lie in the tenths of microsecond range and were performed on the file system cache for a small read and write buffer. This makes it difficult to predict what the effect on a "normal" system is. The results do, however, reflect the worst case scenario, where a program performing rapid and short read or write operations could be slowed down noticeably. For the general case, though, we do not expect this to occur.

Table 4: Network latency tests – times in μs

Kernel	pipe	AF_UNIX	UDP	TCP
FreeBSD	11.2702	12.5225	17.0421	17.8300
Label-0	11.4273	12.6802	17.1144	18.0908
Label-s	11.4340	13.5479	18.0801	18.9715
Label-l	11.4582	13.5722	18.0479	18.9950

The local networking latency times are shown in Table 4. The measured overhead in all instances is small. There is no noticeable difference between the `FreeBSD` and `Label-0` versions, as expected. For `Label-s` the we measured an average overhead of 8.2% for AF_UNIX sockets, 6.1% for UDP, and 6.4% for TCP. `Label-l` has an observed overhead of 8.4% for AF_UNIX, 5.9% for UDP, and 6.5% for TCP. The fact that the measurements for the pipe latency do not vary noticeably between the different versions leads us to believe that pipes on FreeBSD do not utilize the socket subsystem despite such claims [31].

Table 5 shows the I/O bandwidth measurement results. Surprisingly, despite the rather large overhead for the `write` system call test the file write performance does not differ noticeably from that of the original FreeBSD kernel, even outperforming it slightly during our measurements for `Label-s` and `Label-l`. Between the `FreeBSD` and `Label-0` versions we did not observe any noticeable difference. The fact that there is a 2.0% improvement in our measurements for `Label-0` in the TCP throughput, however, suggests that there is a large variance in the network bandwidth tests as can

[4]file write is in KB/s

Table 5: I/O bandwidth tests – in MB/s

Kernel	file write [4]	TCP	AF_UNIX	pipe
FreeBSD	63576.9450	377.8209	617.4526	1845.4127
Label-0	63571.9400	385.6075	612.9818	1842.9119
Label-s	63630.4250	365.5163	544.0799	1835.8787
Label-l	63595.2050	366.4497	543.9735	1833.3753

also be seen in the standard deviation values [5]. `Label-s` shows a deprecation of 3.3% in the TCP throughput and a deprecation of 11.9% for the AF_UNIX bandwidth. For `Label-l` we measured a TCP bandwidth 3.0% worse than `FreeBSD` and an AF_UNIX deprecation of 11.9%. Again, pipes are not noticeably affected by process labeling, which we take as further evidence that they do not use the socket subsystem.

The overall performance overhead that we observed in our experiments for process labeling is promising. The overhead for the individual tests ranges from smaller than 1% to no more than 10% in many cases. The `read` and `write` tests carry a higher overhead of up to 51.9%, but the file write bandwidth was not affected at all. For programs that frequently write small amounts of data into the file system cache this will be a problem, but overall we do not expect the file system slowdown to be severe, especially when factoring in the time to write the cache to the disk.

The network measurements were performed for a label vector size of 128 bytes (1024 labels), and we expect the overhead to be smaller for smaller vector sizes. This may well be the case for certain fixed-label set size applications of a process labeling approach, such as user influence, where we expect the label vectors to be smaller than 10 bytes for most systems. If a fully dynamic approach with a potentially unbounded label set size is desired, however, performance is likely to suffer more.

6 Conclusions

In this paper we presented a label-propagation model and its implementation, that lets a system propagate arbitrary labels of information among its principals and objects based on how information flows within the system. We have demonstrated how those labels can be used to gain some of the desired information regarding the causal effects of events. Our proof-of-concept implementation shows the feasibility of incorporating label propagation for a production-type operating system with little to no overhead.

In previous work [8] we have discussed desired information from a forensic investigator's point of view. For a typical home computer none of the extra information may be worth the space requirements or restrictions that would result from recording it. However, when factors such as due diligence, protecting critical information, or being

able to quickly determine what happened on a computing system are important, all of the extra information discussed may play an important role. Current systems do not offer the ability to record much of the desired information even if one wanted to record it.

The label propagation mechanism we present in this paper can supply some of the lacking information in the form of user influence and location information. The examples we provide in Sections 5.1, 5.2, and 5.3 demonstrate the effectiveness of our approach. We had already demonstrated the value of labeling for network traceback in earlier work [6, 7]. When examining processes and objects on the system we now can make statements as to whether or not they were influenced by external factors such as different users of remote locations. We are not able to assert for sure that if a label is found there was an actual influence, but we have proved in previous work [5] that the lack of such labels means that no communication took place through those channels of the system that support label propagation. Keeping the label sets of the principals and objects of a system small is therefore an important goal, which further needs to be addressed by future research. For this, a combination of label propagation and access control mechanisms could be devised.

Labels are mostly meaningful when their presence at principals and objects is limited. For example, if labels are user identifiers, and a User A's actions were influenced by another malicious User B, then A's process is labeled with both user ids. If only those two labels are present and there is malicious behavior by A the list of potential culprits may be narrowed down to A and B. If, for whatever reason, A is labeled with all possible user identifiers on the system, the labels have become worthless because no information may be gained from them. For this reason, a production system that implements our model should make sure that labels are only propagated when necessary and that a principal cannot add new labels to its label set frivolously to obscure its label set. Our proof-of-concept implementation in Section 4 adheres to these principles. Also, it might be necessary to develop new, label-friendly programming paradigms. These could include the concept of an *execution context* for processes, where a new context is created for a specific task and then discarded without the process gaining information about what occurred while the context was active. This way, labels could be bound to the context and disappear with it instead of remaining with the process even though they are not relevant for any new tasks. This effect of *label creep* needs to be examined further and lies out of the scope of this paper.

For these reasons any framework that utilizes the model should make sure that labels only be propagated and created when absolutely necessary. Well-behaving principals from our case studies should not carry many labels in their label sets. In a computing system, opportunities need to be created for programmers who develop label-friendly programs. If the framework is used to enforce policy, then access control mechanisms need to be implemented in conjunction with the label propagation. This can further reduce the probability that a well-behaving principal acquires unnecessary labels. For example, if a user by default does not have any access to other users' data,

he will only pick up another user label if that user explicitly grants such access and the data is actually accessed. If the framework is used to monitor policy violations, some sort of alert mechanism, such as in intrusion detection, could be utilized to identify principals and objects whose label sets satisfy certain parameters. For this, some sort of human control mechanism that allows an investigation of the label sets and also removal of labels after the investigation, if necessary, may be useful to keep the overall label sets small. For example, after a remote compromise has been detected and contained, the files that were affected by it should be cleared of the labels obtained during the compromise and its consequences. Naturally, such a control mechanism should lie outside the normal system capabilities, such as a special run-level with an operator sitting at the console.

Given this, it becomes clear that a computing system for the average home user is not the target platform for our model. Limitations imposed on the system would be too restrictive to justify the benefits. For example, if peer-to-peer file sharing software were executed for a host causality system, a single file, once completely downloaded, may already have hundreds of labels bound to it. If the user then accesses the file via a shell command, the process running the shell inherits the labels and so will all the subsequent processes that are executed from that shell and the files that are affected by them.

For our proof-of-concept implementation we have chosen an operating system kernel, as the kernel's system call interface to the user space closely resemble the operations described in our model. For label propagation to be secure in this scenario, we have to make the assumption that the kernel is trusted and untampered. Securing the kernel is certainly outside the scope this paper, but project such as LIDS [47] can be used for such measures. An operating system's kernel is not the only place where labeling of principals may be implemented. Integrating label propagation into a virtual machine will enhance the trustworthiness of the labels but comes at the cost of utilizing the virtual machine. This approach is already being realized [23]. But label propagation may also be utilized in user applications or a middle-ware layer. This may be desired when the label granularity of system objects is too coarse to be of value for certain applications. Consider a database management system where the databases are stored in large files. Associating labels about database transactions with the files themselves might result in too many labels being associated with too many entities of the database system. If instead the database management system implemented label propagation through the interface it provides to the clients, labels could be used to analyze or enforce information flow within a database.

7 Limitations and Future Work

The work we present in this paper addresses many important aspects of label propagation, the model's properties, space concerns, and correctness as well as usability. However, there are limitations that we do not specifically address but may be of concern. Some of the limitations we discuss in the following apply to the implementation of the model only, whereas others are also true for the theoretical model.

7.1 Privacy

The label data is not encrypted and potentially can be seen by any user on the system. Depending on the nature of the label, this may lead to privacy concerns. One simple measure could be to limit access to the labels only to a limited set of users. However, this may not be sufficient in all scenarios. One can imagine situations where only the originator of the label should have the option to disclose the data. Using a random token instead of the actual label data may solve the problem in some cases, but once the mapping of token to data is revealed, all users once again have access to the label data. However, because the only requirement we impose on the label update function is that the label needs to be preserved (or at least one should be able to deduce the previous label from the current one), one can imagine the use of cryptographic functions that generate a unique identifier for the result of each update operation, where the original label data may be decrypted with a key. However, we would expect that this functionality would come with a performance cost.

7.2 Label Accumulation

When a principal or an object accumulates many labels, the usefulness of the labels' presence degrades. When a subject possesses all possible labels, an investigator has gained no extra information compared to a system that does not utilize label propagation. This kind of *label obfuscation* may be attempted deliberately by a malicious principal to obfuscate his tracks. In many existing systems there exist globally writable objects that are read by many principals on the system. In FreeBSD there is the `syslog` facility to which all processes may report and is further accessed by many processes on the system. Services such as `cron` and `at` also may pick up labels from all the processes that utilize them. These globally shared objects are a problem when looking to avoid label obfuscation. One could try to eliminate the global sharing by providing mini-services that are valid only for one given process. While this may be acceptable for `cron` and `at`, a per-process `syslog` service would yield scattered log files, with much less usefulness than a single log file would have.

The above discussion about how to deal with label obfuscation and globally shared resources on an existing system shows that label propagation will not always work smoothly together with existing systems. Our proof-of-concept implementation shows

that label propagation can work with a system such as FreeBSD, but some of the limitations will be difficult if not impossible to remove. The intent of our work lies primarily in the introduction of the label propagation paradigm. While legacy systems may be adapted for label propagation, we feel that systems designed with label propagation in mind will benefit the most from the concepts discussed here. But not only systems should be designed with label propagation in mind. *Label-friendly* programming techniques could be used to keep the label set of a process as small as possible. If all "normal" programs adhere to this principle, misbehaving processes could be better identified and contained.

7.3 Multi-threading and multi-processors

The implementation of label propagation we present in this paper does not support separate label sets for threads of a multi-threaded process. As threads share memory space among them, the system is not able to monitor information flow between threads. Given that some programs use a large number of threads to perform tasks, in some cases the granularity of labels for the entire process may be too coarse, meaning that the process will accumulate all of its threads' labels. However, as with shared memory, barring special hardware that can monitor those information transfers, implementing label propagation on the operating system level prohibits label propagation on the thread-level.

We also do not discuss architectures that utilize multiple processors. In this case the sense of a "system" is distributed among the processors, which can lead to problems for label propagation due to concurrency.

7.4 Label retention

Labels are not stored permanently in our proof-of-concept implementation. Once the system reboots, all labels are lost. Incorporating label support directly into a file system, as mentioned in Section 4, is one necessary step to achieve a permanent label retention on a system. In addition to that the labels need to be read from storage at start-up and be written to storage at shutdown. This introduces the danger that labels may not be stored when the system is not shut down properly (i.e. it crashes). Storing the label periodically can reduce the number of labels that are potentially lost, but the threat of losing labels remains. Also, if labels sets are stored in label-vectors as in our proof-of-concept implementation, the global label table needs to keep labels always in the same order. This is because not all labels are necessarily introduced to the system at start-up. Some file systems with labels may be mounted at a later time at which point the label-vector bits need to point to the correct entries in the label table. Furthermore, if a file system from a different system is mounted and also has labels associated with it, the new labels need to be incorporated into the current system or be discarded. In the former case, the label data has to be stored on the file

system device, as well.

We see as immediate topics of future work the following list:

- Research if and how existing programs and programming paradigms fit in with a label propagation framework in regard to the amount of labels they accumulate and the implications this has for access control.
- Determine how to integrate label propagation into access control and intrusion detection mechanisms.
- Explore new uses for labels outside of those discussed in this paper.
- Identify scenarios where label propagation and its (possible) limitations award the most benefits to justify its use.

References

[1] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. *MITRE Report MTR 2547 v2*, 1973.

[2] K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, MITRE Corporation, Bedford, MA, 1977.

[3] D. Brewer and M. Nash. The Chinese Wall Security Policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.

[4] Florian Buchholz. The Structure of the Reiser File System. `http://www.cerias.purdue.edu/homes/florian/reiser/reiserfs.php`.

[5] Florian Buchholz. *Pervasive Binding of Labels to System Processes*. PhD thesis, Purdue University, West Lafayette, IN, August 2005.

[6] Florian Buchholz and Clay Shields. Providing Process Origin Information to Aid in Network Traceback. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, CA, July 2002. CERIAS TR 2002-22.

[7] Florian Buchholz and Clay Shields. Providing Process Origin Information to Aid in Computer Forensic Investigations. *Journal of Computer Security*, 12(5):753–776, September 2004.

[8] Florian Buchholz and Eugene H. Spafford. On the Role of File System Metadata in Digital Forensics. *Journal of Digital Investigation*, 1(4):298–309, December 2004.

[9] Florian Buchholz and Eugene H. Spafford. Run-time label propagation for forensic audit data. Technical Report INFOSEC-TR-2006-001, Department of Computer Science, James Madison University, 2006.

[10] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In Frank B. Brokken et al., editor, *Proceedings of the First Dutch International Symposium on Linux*, 1994.

[11] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.

[12] Dorothy E. Denning. Secure Personal Computing in an Insecure Network. *Communications of the ACM*, 22(8):476–482, 1979.

[13] Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

[14] Dorothy E. Denning and Peter F. MacDoran. Location-based Authentication: Grounding Cyberspace for Better Security. In *Internet Besieged: Countering Cyberspace Scofflaws*, pages 167–174. ACM Press/Addison-Wesley Publishing Co., 1998.

[15] G.W. Dunlap, S.T. King, S. Cinar, M.A. Basrai, and P.M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-machine Logging and Replay. In *5th Symposium on Operating Systems Design and Implementation*, December 2002.

[16] J.S. Fenton. Memoryless Subsystems. *The Computer Journal*, 17(2), 1974.

[17] FreeBSD Operating System. `http://www.freebsd.org`.

[18] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *2003 Symposium on Network and Distributed System Security*, February 2003.

[19] J.A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 11–20, Oakland, CA, 1982.

[20] Nevin Heintze and Jon G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[21] Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Secure Information Flow as Typed Process Behaviour. *Lecture Notes in Computer Science*, 1782:180–199, 2000.

[22] IEEE Standard for Information Technology - Portable Operating System Interface (POSIX). `http://standards.ieee.org/catalog/olis/arch_posix.html`.

[23] Xuxian Jiang, Dongyan Xu, and Florian Buchholz. Tracking Worm Contamination: a Process Coloring Approach. *under submission*, May 2005.

[24] R. Joshi, K. Rustan, and M. Leino. A Semantic Approach to Secure Information Flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[25] Gene H. Kim and Eugene H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[26] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 223–236. ACM Press, 2003.

[27] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-traveling Virtual Machines. In *Proceedings of the 2005 Annual USENIX Technical Conference*, April 2005.

[28] Benjamin A. Kuperman. *A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources.* PhD thesis, Purdue University, West Lafayette, IN, 08 2004. CERIAS TR 2004-26.

[29] B. Lampson. Protection. In *Proc. 5th Princeton Conf. on Information Sciences and Systems*, Princeton, 1971. Reprinted in ACM Operating Systems Rev. 8, 1 (Jan. 1974), pp 18-24.

[30] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

[31] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System.* Addison Wesley, Boston, MA, 1996.

[32] John McLean. Proving Noninterference and Functional Correctness Using Traces. *Journal of Computer Security*, 1(1), 1992.

[33] L. McVoy and C. Staelin. LMBench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, January 1996.

[34] L.W. McVoy and S.R. Kleiman. Extent-like Performance from a Unix File System. In *USENIX Winter Conference*, pages 33–43, January 1991.

[35] J. Palsberg and P. Ørbæk. Trust in the Lambda-calculus. *Journal of Functional Programming*, 7(6):557–591, 1997.

[36] A. Sabelfeld and A. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[37] Geoffrey Smith and Dennis Volpano. Secure Information Flow in a Multi-Threaded Imperative Language. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 355–364, New York, NY, 1998.

[38] W.R. Stevens. *Unix Network Programming*, volume 1. Prentice Hall PTR, Upper Saddle River, NJ, second edition, 1998.

[39] W.R. Stevens. *UNIX Network Programming: Interprocess Communications*, volume 2. Prentice Hall PTR, Upper Saddle River, NJ, second edition, 1999.

[40] Sun Microsystems. SunSHIELD Basic Security Module Guide. `http://docs.sun.com/db/doc/802-5757`.

[41] Wietse Venema. TCP WRAPPER, A Tool for Network Monitoring, Access Control, and for Setting Up Booby Traps. In *Prococeedings of the 1992 USENIX Security Symposium*, September 1992.

[42] D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. In *Proceedings of The 11th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[43] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[44] Christian Wettergren. Runtime Information Flow Analysis and Security: Licentiate Thesis Proposal. `http://www.it.kth.se/~cwe/phd/licprop.ps`, 1996.

[45] A. Whitaker, R.S. Cox, and S.D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of USENIX OSDI 2004*, December 2004.

[46] A. Whitaker, R.S. Cox, and S.D. Gribble. Using Time Travel to Diagnose Computer Problems, September 2004.

[47] Huagang Xie and Philippe Biondi. Linux Intrusion Detection System. `http://www.lids.org`.

[48] Diego Zamboni. *Using Internal Sensors for Computer Intrusion Detection*. PhD thesis, Purdue University, 2001. CERIAS TR 2001-42.