Towards More Usable Information Flow Policies for Contemporary Operating Systems

Wai Kit Sze, Bhuvan Mital and R. Sekar
Stony Brook University
Stony Brook, NY, USA

ABSTRACT

There has been a resurgence of interest in information flow based techniques in security. A key attraction of these techniques is that they can provide strong, principled protection against malware, regardless of its sophistication. In spite of this advantage, most advances in information flow control have not been adopted in mainstream operating systems since a strict application of information flow can limit system functionality and usability. Permitting dynamic changes to subject labels, as proposed in the low-watermark model, provides better usability. However, it suffers from the *self-revocation* problem, whereby read/write operations on already open files are denied because the label of the subject performing these operations has been downgraded. While most applications deal gracefully with security failures on file open operations, they are unprepared to handle security violations on subsequent reads/writes. As a result, subject downgrades may lead to crashes or malfunction. Even those applications that deal with read/write errors may still leave output files in a corrupted or inconsistent state since write permissions were taken away in the midst of producing an output file. To overcome these drawbacks, we propose a new approach for dynamic downgrading that eliminates the self-revocation problem. We show that our approach represents an optimal combination of functionality and compatibility. Our experimental evaluation shows that our approach is efficient, incurring an overhead of a few percentage points, is compatible with existing applications, and provides strong integrity protection.

1. Introduction

Operation Aurora [2] and Stuxnet [7] signified the arrival of an era of targeted attacks by sophisticated malware. Their pace has only quickened in the past two years, as malware attack campaigns are revealed on major organizations at alarmingly regular intervals.

Security in contemporary operating systems such as Windows and various flavors of UNIX is based on discretionary access control (DAC) that relies on userids. It suffers from the well-known weakness that DAC cannot distinguish a malicious user from a malicious program. As a result, if a legitimate user happens to run a malicious program, this program can co-opt all of the user's privileges to defeat or circumvent system security. Worse, malware need not even be executed by the user in most cases, as it can take the form of malicious data that hijacks a vulnerable benign application. According to Trend Micro [29], at least 70% of targeted malware attacks compromised victim systems using non-executable content such as PDF or JPEG.

The weaknesses of userid-based DAC has prompted a resurgence of interest in mandatory access control (MAC) [17, 4, 13, 26, 30, 6, 5, 14, 18, 28, 8]. Information-flow approaches such as the Biba model [4] are particularly attractive in the context of malware threats, as they can prevent low-integrity (untrusted and potentially malicious) data or code from ever influencing high-integrity data or applications. It not only prevents malware from directly corrupting important system files, but also stops indirect attacks that operate by corrupting some intermediate data consumed by other high integrity processes that can update important system files.

A drawback of the Biba model is that its strict separation between high and low-integrity objects and subjects, which impacts its usability. Consider a utility application such as a word-processor that needs to operate on both high and low integrity files. It would be necessary to have two versions of every such application, one for operating on high-integrity files and another for low-integrity files. It is cumbersome to install and maintain two versions of every application. Worse, a user needs to be careful in selecting the correct version of an application for each task — choosing a high-integrity version of an application for processing low-integrity files (or vice-versa) will lead to security failures and/or application crashes.

The low-watermark policy [4] can avoid these drawbacks of the strict policy by permitting subject integrity to be downgraded at runtime. In particular, this policy allows applications to be invoked with high integrity, and the integrity level to be downgraded if the application subsequently reads a low integrity object. Fraser [8] argues eloquently why low watermark policy provides significantly better compatibility with existing software as compared to the strict model. However, prior to this project, the low watermark policy was not very popular because of the *self-revocation* problem [8]. Specifically, consider a subject that has already opened a high integrity file for writing. If this subject subsequently opens a low integrity file for reading, it is downgraded. At this point, the subject cannot be permitted to write the high integrity file any more. Applications expect and handle security failures when opening files, but once opened, they assume that subsequent read and write operations will not fail. When this assumption is invalidated, applications may malfunction or crash.

The LOMAC project [8] does not attempt to solve the self-revocation problem in its entirety, but focuses on two common instances

SACMAT'14, June 25–27, 2014, London, Ontario, Canada.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2939-2/14/06 ...$15.00.
http://dx.doi.org/10.1145/2613087.2613110.

†This work was supported in part by grants from NSF (CNS-0831298, CNS-1319137) and AFOSR (FA9550-09-1-0539).

that involve pipes and shared memory abstractions. Pipes are particularly nasty, because downgrading of one process in a pipeline can prevent it from writing to the pipe, which in turn will cause the next process in the pipeline to fail because it does not get any input. LOMAC avoids this problem by permitting pipe communications only within a UNIX process group, and ensuring that all processes within this group are at the same level. This notion of a group is further extended to include all processes that share memory. Since all processes within a group are at the same level at all times, there is no need to restrict communication among them, and hence pipes and shared memory operations don't ever have to be denied. Unfortunately, self-revocation problem still remains when dealing with files, as well as other IPC mechanisms such as sockets.

In this paper, we propose a more general solution to the self-revocation problem in all cases. Our specific contributions are:

- We develop a model (Section 2) to compare different integrity policies in terms of their functionality (i.e., the behaviors that they permit) and failure compatibility (i.e., ability to map security failures newly introduced by the policy into those that are already handled by an application). We show that among these policies, low-watermark policy is the best in terms of functionality, but the worst in terms of compatibility.
- We then define a new policy, called *Self-Revocation-Free Dynamic Downgrading (SRFD)* that combines the best features of different information flow policies.
- We present a design for enforcing SRFD on contemporary operating systems. (See Section 3.) Our design uses a novel constraint propagation technique to identify file open operations that introduce a potential for future self-revocations, and denies them. Our design is general, and avoids self-revocation involving files as well as interprocess communication.
- We formally show that SRFD eliminates self-revocations. We also show that unless future inter-process communications can be predicted accurately, it is not possible to improve on the functionality of SRFD without incurring self-revocation.
- We present an implementation of SRFD on Ubuntu Linux 13.10 in Section 4. Our experimental evaluation (Section 5) shows that our implementation is fast, incurring a maximum overhead under 6% and average overhead below 2% across several macro-benchmarks. The evaluation also demonstrates that SRFD provides very good compatibility, while thwarting malware attacks.

An open source implementation of SRFD can be found at [27].

2. Model

In this section, we consider information-flow policies that are commonly used in the context of integrity preservation: strict integrity policy and (two flavors of) dynamic downgrading policy. We propose a model to compare these security policies. Specifically, we show that a policy that supports dynamic downgrading of subjects provides better functionality than the strict integrity model. However, more functionality does not always translate to better compatibility or user experience. We therefore formalize the notion of compatibility, and proceed to define a new dynamic downgrading policy that provides an optimal combination of functionality and compatibility among the commonly-used integrity policies.

We model process execution in terms of the sequence of actions $\mathbf{A} = A_1 \ldots A_n$ performed by a process. Each action A_i can be:

- *an invocation (I),* typically, the execution of another program;
- *an observation (O),* typically, a file read operation; or
- *a modification (W),* typically, a file write operation.

In order to simplify terminology and description, we consider only two integrity levels in this paper: high (Hi) and low (Lo). Objects (typically files) as well as subjects (processes) have one of these integrity levels.

DEFINITION 1 (INTEGRITY-PRESERVING EXECUTIONS). *Such executions ensure that the content of all high integrity objects are derived entirely from other high integrity objects and subjects.*

A strong integrity-preservation policy, such as the Biba's strict integrity policy and the low-watermark policy, will ensure that all executions are integrity preserving. In particular, this means that low-integrity data and programs cannot influence the contents of integrity-critical (data or program) files on the system. Today's remote exploits and malware attacks all rely on modifying critical files using data or code from untrusted sources, and hence can be definitively blocked by enforcing these integrity policies, provided we ensure that only data/code from trustworthy sources is given a high integrity label.

When a security policy is enforced, it can alter an execution sequence in one of two ways. First, it can disallow an operation A_i, denoted as $\not{A}_i$. There are several possibilities here, including (a) silent suppression of A_i, (b) suppressing A_i and returning an error to the process performing this operation, and (c) replacing A_i with another allowable action. In the rest of this paper, we primarily focus on the alternative (b).

A second avenue for the enforcement engine is to downgrade a subject before A_i, denoted $\downarrow A_i$. Note that such a downgrade may be an internal operation within a reference monitor enforcing the policy, and hence we may not explicitly show it in some instances.

Executions without any failed operations are called *permitted* or *successful executions,* while the rest are called *failed executions.* The more execution sequences that a security policy permits, the less functionality will be lost as a result of security policy enforcement. This leads to the following definition comparing the functionality supported by security policies.

DEFINITION 2 (FUNCTIONALITY). *A security policy P_1 is said to be more functional than P_2, denoted $P_1 \supseteq_F P_2$, if and only if every execution sequence permitted by P_2 is also permitted by P_1.*

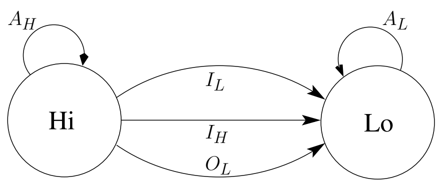
Note that functionality defines a partial order on security policies, and hence two policies could be incomparable in terms of functionality. By permitting more executions, a more functional policy would seem to provide weaker security than a less functional policy, thus capturing the tension between functionality and security.

2.1 Integrity policies

We can now classify actions into two categories: *high integrity actions* (A_H) that can be performed by high integrity subjects, and *low integrity actions* (A_L) that can be performed by low integrity subjects. Specifically, A_H includes all actions except read-down (O_L), i.e., read a low-integrity input, and invoke-down (I_L), i.e., executing a program that has low integrity. A_L includes all actions except write-up (W_H). Note that I_H is permitted in A_L because we interpret it as the execution of a high integrity file within a low integrity subject. (In contrast, the term "invoke-up" is used in Biba model to refer to the execution of a high integrity subject.)

Integrity-preserving execution sequences can be achieved by confining high integrity processes to perform only A_H, and low integrity processes to perform only A_L. Since W_H exists only in A_H and O_L exists only in A_L, it is clear that low-integrity objects and subjects cannot affect high-integrity objects.

Since we want to protect the integrity of critical files, revisions to to object integrity levels are disallowed in most systems. However, subject integrity label can be revised down as long as the down-



$$A_H = \{O_H, W_H, W_L, I_H\}$$
$$A_L = \{O_H, O_L, W_L, I_L\}$$

Figure 1: State machine for integrity-preserving executions.

graded subject is restricted to perform A_L after the downgrade. This leads to the following variants that all preserve integrity.

No Downgrading (ND). This policy, which corresponds to the Biba strict policy, permits no privilege revision (NPR): labels are statically assigned to subjects and objects, and they cannot change. With this strict interpretation, every program has to be labeled as high or low integrity, and a high integrity program cannot be used to process low integrity data, even if all outputs resulting from such use flow only to low-integrity files or low-integrity subjects.

Eager downgrading (ED). This policy permits subject labels to be downgraded, but only when executing another process. This approach, also called *privilege revision on invocation* (PRI), allows more executions as compared to the no-downgrading policy. With the PRI policy, a subject wishing to operate on low-integrity files should know ahead of time (i.e., prior to execution) that it needs to consume low-integrity file, and drop its privilege before execution. This is why we call it *eager downgrading.*

Lazy downgrading (LD). The final policy is the low watermark policy for subjects, where downgrades can happen before any observe operation, or an invoke operation. We call it lazy (or just-in-time) downgrading since downgrading operation would typically be delayed until the very last step, which must be the consumption of a low-integrity input.

Figure 1 shows a simple state-machine model that captures the above three policies. With the ND policy, none of the transitions between H and L states are available. With the ED policy, only the transitions on I_L and I_H are enabled. Note that while it is mandatory to transition to Lo on I_L, I_H may or may not cause a transition to Lo. When it does, it corresponds to the use of a high-integrity application to process low-integrity data.

With the LD policy, only the I_L and O_L transitions from Hi to Lo are enabled. There is no need to make a transition from Hi to Lo on I_H, as the downgrade can be deferred until the next operation to read low-integrity data. As a result, LD avoids one of the difficulties of ED, namely, the need to predict ahead of time whether a certain process will need to read low-integrity data.

2.2 Comparing functionalities of integrity policies

It is easy to see the motivation for the LD policy: when the actions performed by an application are disallowed, it can lead to errors and failures, and hence loss of functionality. In contrast, downgrading has the potential to permit the application to continue to provide its function. In fact, we can formally state:

THEOREM 3. $LD \supset_F ED \supset_F ND$

Proof: This theorem simply states that LD is strictly more functional than ED, which, in turn, is more functional than ND. From the definition of the three policies, and Figure 1, it is easy to see that all three policies accept the same set A_L^* of execution sequences for low-integrity subjects. (We are using a regular expression syntax to succinctly capture the set of execution sequences permitted by a policy.) Thus, we can limit our comparisons to the execution sequences permitted for high-integrity subjects. Note that ND accepts only sequences of the form A_H^* for high subjects. ED accepts $(I_L|I_H)A_L^*$ for subjects started with high, in addition to the set A_H^*. Finally, LD accepts $A_H^*(I_L|O_L)A_L^*$, which is a strict superset of sequences accepted by ED. ■

2.3 Compatibility

Increased functionality does not always translate to a better user experience, or better compatibility with existing software. Self-revocation is a prime example of the compatibility problem posed by LD, an approach that maximizes functionality over other integrity policies. In contrast, ED provides less functionality as compared to LD, but is intuitively perceived as being more compatible.

Self-revocation occurs when a subject is initially granted access to a resource, but this access is revoked subsequently; and the revocation is the result of some of the other actions performed by the subject itself. More concretely, self-revocation manifests itself as follows in the context of file system APIs provided by modern operating systems: a process successfully opens a file, but a subsequent write operation using that file handle is denied. Although self-revocation is more commonly identified with failures of writes, it can also happen on read operations. In both cases, self-revocation raises several compatibility issues:

- The file system API is designed to perform security checks on open operations, but not on reads and writes. As a result, there is usually no way to even communicate a security failure to the subject performing the read or write[1]. Thus, security failures have to be mapped into other failures that can occur on reads/writes, such as an attempt to read a file before opening it. Such remapping has obvious drawbacks because applications may misinterpret the error code and respond inappropriately.
- Even if an error code is returned on reads and writes, many applications may not check them at all. This is because failures of these operations are rare and unexpected, so many applications may not contain code for checking these error cases, or undertaking any meaningful error recovery.
- Even if the application checks the error and undertakes recovery, data loss or corruption may be unavoidable at this point. Consider an application that was updating a file. If its write access is taken away when it is half-way through the update, that may lead to the file being truncated, leading to data loss, inconsistency or corruption.[2]

For this reason, we develop the following notion of failure-compatibility, or simply, compatibility of security policies.

DEFINITION 4 (COMPATIBILITY). *We say that a security policy P is compatible if all actions disallowed by it can return a valid permission failure error to the subject.*

With contemporary file APIs, this means that a compatible policy would deny `open`'s but not reads/writes. We show that in terms of compatibility, the results are inverted from that of functionality:

THEOREM 5. *LD is not failure-compatible, whereas ND and ED are both failure compatible.*

Proof: Recall that for subjects that start at low integrity, all three policies allows A_L^*. It is clear that this sequence permits the same

[1]For instance, on UNIX, there are no error codes related to permissions that can be returned by read and write system calls.

[2]With buffered I/O, even the data that an application believes to have written prior to the self-revoking action may be lost — such data may be held in the program's internal buffers, which may be flushed much later, at which point, the write system call would fail.

set of operations throughout, so self-revocation is not possible. For high integrity subjects, ND accepts A_H^* — again, the set of operations permitted remain constant throughout the subject's lifetime, and hence there will be no self-revocation. For ED, the sequences accepted are of the form A_H^* or A_L^*. For each alternate, it is easy to see that all of the actions permitted towards the beginning of the sequence are also permitted later on, once again ruling out the possibility of self-revocation. Finally, we have already explained how LD suffers from self-revocation. ■

2.4 Maximizing functionality *and* compatibility

The results above lead to the following question: can there be an approach that is preferable in terms of both functionality and compatibility? Our answer in this paper is affirmative. We begin by positing the existence of a new dynamic downgrading policy that combines LD's functionality with the compatibility of ED.

DEFINITION 6 (SELF-REVOCATION-FREE DOWNGRADING). *SRFD accepts the same set of execution sequences as LD. Every sequence that is modified by LD is also modified by SRFD, but unlike LD, SRFD only modifies (i.e., denies) open operations.*

So, the next natural question is whether SRFD is realizable. Conceptually, we can synthesize execution sequences accepted/modified by SRFD from the acceptance and modification actions of LD as follows. If LD accepts a sequence, then SRFD will accept the same sequence. If LD modifies a sequence, let A_i be the first write operation denied by LD. SRFD will identify the open operation A_j preceding A_i that caused LD to downgrade the subject, and then SRFD will deny A_j.

Noting that LD denies only write operations on high-integrity files, this means that SRFD needs to predict whether a subject will perform future writes on any of the currently open file descriptors for accessing high-integrity files. If so, SRFD should not permit the subject to open any low-integrity file. In this manner, SRFD can prevent the subject from downgrading itself, and hence will not have to deny writes on one of these descriptors in the future.

This raises the final question: how can a reference monitor predict future actions of a subject? Often, questions regarding future behavior are answered by assuming that any thing that can happen will indeed happen. We formalize this by characterizing a class of programs that transfer data along every possible communication channel between communicating processes, and show that for this class, SRFD can indeed be realized.

Another way to characterize our result is as follows. Unless an oracle for predicting future behavior of a set of communicating processes exists, one cannot improve over the functionality of the design presented in the next section without risking self-revocation.

3. Our approach

Our approach represents a hybrid between ND that refuses to ever downgrade a subject, and LD which downgrades at the first open of a low-integrity file. The key idea is to deny these open operations when a subject already holds open file descriptors that write to high-integrity files. This task is simple enough for stand-alone subjects, but challenges arise when considering processes that interact with each other.

Note that many applications involve processes that communicate via pipes, sockets, shared memory and other IPC mechanisms. If we look at each process in isolation and allow one of them to be downgraded, it is possible that a future read by another process would have to be denied, since it is reading an output of the downgraded process. Since the goal of our approach is to avoid denials of reads/writes, it would seem that we need better mechanisms to keep track of open file descriptors across collections of processes.

A simple approach to deal with collections of communicating processes is to treat them as a single unit, and downgrade them as a unit. LOMAC [8] uses this approach to avoid self-revocation due to IPC within a UNIX process group. However, this approach does not recognize the one-way nature of pipe-based communication, and hence would needlessly downgrade an upstream process when a downstream process opens a low-integrity file. To avoid this, it would seem that we need a mechanism to keep track of all output files held open by processes that are downstream from each process. Since this information is different for each process, keeping track of it can be messy as well as expensive, especially if the number of processes (or number of open files) grows large.

To overcome these problems, we develop a new approach that is based on propagating constraints about downgradability of processes. In particular, we keep track of the highest integrity of any output file that is held open by a process and any of the processes that it writes to. We call this `min_lbl` and propagate it "upstream" through pipes and other communication mechanisms. The result is an approach that relies on maintaining/propagating just this single quantity (`min_lbl`) for each process, instead of having to propagate a large amount of information concerning open file descriptors.

We now proceed to describe the key abstractions in our design and our constraint propagation mechanism. Although we have limited ourselves to just two integrity levels, the design described below is quite general and can support any lattice of integrity labels. While our design is fully compatible with unmodified COTS applications, it does provide features that can be utilized by information-flow-aware applications to provide improved functionality. One such feature enables an application to explicitly request that it not be downgraded below a certain level. In particular, this means that any attempt to open files at a lower integrity than this specified level should be denied. Another feature allows trusted applications[3] to request selective, fine-grained exceptions to the information-flow policy. Although we do not discuss these features in depth in this paper due to space constraints, we point out that fully working systems need a handful of key administrative and helper applications that rely on these features.

3.1 Abstractions

Our design uses three entities: Objects, Subjects and Handles.

Objects. Objects consist of all storage and inter-process communication abstractions on an OS: files, pipes, sockets, message queues, semaphores, etc. These objects are divided into two categories: file-like and pipe-like. There is a fundamental difference between these classes. File-like objects are persistent, and have a fixed label assigned to them. Any data read from the file has this label, and writes to the file don't change the label. (The information flow policy ensures that any subject writing to it has a equal or higher label.) For a file-like object, the label of data read from it will be the same as that of data written into it. In contrast, for a pipe-like object, the label of data read from the object representing one end of the pipe is the same as the label of data written to the object representing the other end of the pipe (called a peer object). Examples of pipe-like objects include UNIX pipes and sockets.

Subjects and SubjectGroups. Subjects correspond to threads. Since the OS-level mechanisms used in our framework cannot mediate information flows that take place via shared memory, subjects

[3]These are applications that have been written carefully so as to protect themselves from low-integrity data, and hence can operate on them while retaining their high integrity.

that share memory are grouped into SubjectGroups. The idea is that all subjects within a SubjectGroup will have the same security label at any time.

Handles. Handles provide a level of indirection between subjects and objects. They serve to link together objects and subjects that have an information flow relationship. There is a many-to-one mapping between handles and subjects, and many-to-one mapping between handles and objects.

Handles are conceptually similar to file descriptors, but there are some differences as well, e.g., a handle is unidirectional: a handle provides either a read or a write capability. (Obtaining both requires two handles.) The label of a read-handle is given by the label of the object that it reads from, while the label of a write-handle is given by the label of the subject holding the handle. When read (or write) operation takes place, the label of the handle will be passed to the corresponding subject (or object).

3.2 Information Flow Policies

A current label (`current_lbl`) field is associated with each object and subject, and it provides the basis for policy enforcement. In particular, no flow will be permitted from a source to a destination unless the source's current label is greater than or equal to that of the destination.

INVARIANT 7. *Any information flow from an entity A to another entity B must satisfy* $\texttt{current\_lbl}(A) \geq \texttt{current\_lbl}(B)$.

Instead of denying the operation when the above invariant does not hold, our system will attempt to dynamically downgrade the label of the destination. Since the model presented so far restricts downgrading to subjects, B must be a subject, and downgrade occurs when it reads from a handle A. B can protect itself from undesirable downgrades by setting its minimum label (called `min_lbl`). In particular, downgrading of `current_lbl` won't be attempted unless the following invariant holds after the downgrade:

INVARIANT 8. $\forall B, \texttt{current\_lbl}(B) \geq \texttt{min\_lbl}(B)$.

Since we do not downgrade the labels of file-like objects, their `min_lbl` will be the same as their `current_lbl`. For subjects and pipe-like objects, `min_lbl` is determined by constraint propagation, as described further in Section 3.4. Finally, handles do not have an independent value for their current label and minimum label; instead, these are derived from the corresponding values of objects and subjects associated with a handle.

Combining the above two invariants, we can say that our approach will permit information flow from A to B in all cases where $\texttt{current\_lbl}(A) \geq \texttt{min\_lbl}(B)$. Since self-revocation occurs precisely when such a data transfer is denied, we can say:

OBSERVATION 9. *A read (or write) operation that transfers data from an entity A to another entity B will be denied in our approach only if* $\texttt{current\_lbl}(A) < \texttt{min\_lbl}(B)$.

3.3 Forward information flows

Figure 2 illustrates the flow of information between objects and subjects via handles. In this figure, solid lines represent actual flow of information. There are two subjects S_1 and S_2. Flow of information between these two subjects occurs via a socket object O_1 (which is pipe- like), and a file object O_2.

Flow of information via file objects is simpler than that of pipe-like objects. In particular, an object created by a subject receives the label of that subject. This flow is handled by propagating the current label of subject S_2 to its write handle WH_2, and then from WH_2 to the object O_2. (If the object is already present, then its `current_lbl` should be less than or equal to that of the subject writing to it, and no propagation would be needed.) If S_1 subsequently reads from the object O_2, the label of O_2 will flow into S_1.

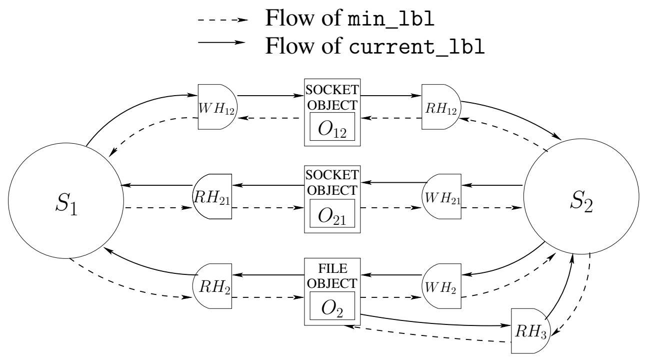


Figure 2: Illustration of Information Flow in our Framework

Since a socket is a pipe-like object representing two distinct flows, we split it into two objects: O_{12} that represents information flow from S_1 to S_2, and O_{21} that represents the information flow from S_2 to S_1. S_1 uses a read-handle RH_{21} and a write-handle WH_{12} to read from and write into the socket, while S_2 uses RH_{12} and WH_{21} respectively for the same purpose.

It is important to clarify the role of `open` versus `read` operations. Specifically, when a file is opened for reading, the file's `current_lbl` flows from the file to the handle. But since no data has yet been read by the subject, the propagation of `current_lbl` from the handle to the subject does not take place until the first `read` operation. (A similar comment applies to `write` operations as well.) This distinction between `open` and `read` operations is made for pipe-like objects as well, except that there are many `open`-like system calls, including `pipe`, `connect` and `accept`.

Delaying `current_lbl` propagation serves an important purpose: shells (e.g., bash) often open files for file redirection, and set up pipes for use by its child processes. The shell process does not perform any reads/writes on these objects. By deferring any downgrades until the first `read`, we prevent the shell from having to downgrade itself. Such a downgrade of shell's label is disastrous, as it prevents the shell from ever running high-integrity commands.

We note that for memory-mapped files, reads may happen implicitly when memory is read, and hence we don't support delayed propagation of labels as described above.

3.4 Constraint propagation

As noted earlier, self-revocation is avoided in our approach by propagating constraints on `min_lbl`. Figure 2 shows constraint propagation using dashed lines. Note that constraints propagate in the reverse direction of information flow.

Note that `min_lbl` represents the minimum label that needs to be maintained by a subject A. Any entity B from which information can flow to A needs to maintain a label higher than $\texttt{min\_lbl}(A)$ or else the flow from B to A may have to be cut-off. Since such cut-offs lead to self-revocation, we want to prevent them. This is accomplished by propagating $\texttt{min\_lbl}(A)$ to any handle from which A reads; and from this handle to the associated object; and so on. In other words, by propagating `min_lbl` in the inverse direction of information flow, we can ensure that every data producer upstream will be able ensure the integrity level required by A.

Whereas the forward flow of labels is normally delayed until an explicit read or write operation, constraint propagation is instantaneous, i.e., when a channel (representing file or pipe-like communication) for information flow from entity A to another entity B is opened, B's `min_lbl` is propagated immediately to A. Because of

Invariant 8, this propagation will fail if A's current label is already less than min_lbl(B). In this case, the open operation is denied.

It is important to note that min_lbl is a quantity that is derived through constraint propagation. It should not be thought of as a variable whose value is increased each time a new communication channel is established. For this reason, min_lbl can either *increase* or *decrease* during the lifetime of a subject. Increases happen when a subject opens a new output handle, while decreases happen when a subject closes an output handle.

Due to constraint propagation, the following invariant holds:

INVARIANT 10. *If there is an information flow path (shown by solid lines in Figure 2) from A to B,* min_lbl$(A) \geq$ min_lbl(B).

Since constraint propagation increases a min_lbl value for an entity only if there is a constraint that requires it to be that high, and since files are the only entities that have a hard requirement for their min_lbl values, we can make the following observation:

OBSERVATION 11. *For an entity A, let $B_1, ..., B_k$ be all the open output files reachable from A while following the information flow paths. Then* min_lbl(A) *will be the maximum among* min_lbl$(B_1), ...,$ min_lbl(B_k).

This observation follows readily from our declarative definition of constraints and their propagation.

3.5 Properties

THEOREM 12. *There will be no self-revocations in our approach.*

Proof: The proof is by contradiction. Suppose that a self-revocation takes place on a read or write operation that transfers data from A to B. From Observation 9, self-revocation can happen only when current_lbl$(A) <$ min_lbl(B). Together with Invariant 8, this implies that min_lbl$(A) <$ min_lbl(B). However, note that it is invalid to issue a read or write operation before setting up the information flow path between A and B. (In this case, the path happens to be of length 1.) From Invariant 10, the condition min_lbl$(A) \geq$ min_lbl(B) must also hold, thus leading to a contradiction. ■

DEFINITION 13 (FLOW-INDETERMINATE PROGRAMS). *A set of programs are said to be flow-indeterminate if for any set of communicating processes running them, the following condition holds: for every communication path p between any two processes, there will be data transfer operations that cause data to flow from the beginning to the end of this path.*

Flow-indeterminacy simply formalizes the idea that programs may exhibit any possible pattern of communication that is consistent with their current set of open file descriptors; and that there is no simple yet general way to delineate likely communications from those that are unlikely/impossible.

THEOREM 14. *For flow-indeterminate programs, any policy that accepts any execution rejected by SRFD will suffer from self-revocation.*

Proof: For an execution sequence rejected by our approach, consider the first operation A_i that is denied. From the description of the approach in Section 3.4, A_i must be an open operation that would have created a path from entity A to B such that current_lbl$(A) <$ min_lbl(B). Now, suppose that there exists a correct integrity policy P that permits this open operation. Then, because of the properties of flow-indeterminate programs, it can be seen that there will be a subsequent operation that transfers data from A to B. This will either have to be denied, or it will cause current_lbl(B) to fall below min_lbl(B). The former case corresponds to self-revocation, thus completing the proof. In the latter case, from Observation 11, it can be seen that there is some output file B_i whose min_lbl is higher than current_lbl(B). Also, from properties of flow-indeterminate programs, there will be an actual data flow from B to B_i, which will cause the output file B_i's label to fall below its minimum value. This is not permissible in the model, and hence the more permissive policy P is simply invalid. Thus, in either case, we have established that the functionality offered by SRFD cannot be increased without risking self-revocation. ■

Thus, for flow-indeterminate programs, we have shown that our approach allows the same successful executions as any other valid information-flow policy that is free of self-revocations. Thus, it represents the maximal functionality achievable without any self-revocations.

4. Implementation

We have implemented SRFD design as described in the previous section. Our implementation uses Linux Security Module (LSM) framework, and works on Ubuntu 13.10. Our code is primarily in the form of handlers for various LSM hooks. Although Linux kernel no longer allows loadable modules to use LSM hooks, there are work-arounds available [1] that we relied on. Structuring the system as a loadable module eases development and debugging, especially in the early stages of prototype development.

LSM hooks are used to enforce information flow policies, perform dynamic downgrading, and to track and maintain min_lbl constraints. Our implementation also uses an user-level component to provide some usability enhancing features, e.g., notifying users when a process is downgraded and shadowing accesses to preference files for low integrity processes. By maintaining separate preference files for high and low integrity processes, SRFD prevents processes from downgrading automatically due to consuming low integrity preference files. Note that these features do not allow a process to bypass kernel enforcement.

The overall size of our implementation is shown in Figure 3.

	C	Header	Python	Total
Kernel Code	3844	865	-	4709
Userland code	643	142	57	842
Total	4487	1007	57	5561

Figure 3: Implementation size

4.1 Subjects, Objects, and Handles

SRFD maps threads to subjects. Threads of the same process belong to the same subject group. Within the kernel, these correspond to task_structs. Since LSM does not provide hooks to track process creation directly, our prototype relies on cred_* hooks instead. For each subject group, SRFD maintains information such as integrity level and a list of handles.

Objects are mapped into inodes in the kernel. Our implementation maintains and updates object-related information, including labels, handles associated with each object, and constraints. LSM hooks on inodes are used for creating objects on demand, and deallocating objects when they are no longer needed. For file objects, integrity labels are stored on the disk using extended attributes provided by the file system.

Handles are similar to file descriptors but represent an unidirectional information flow between exactly one subject and one object. SRFD relies on LSM hooks such as file_open, inode_permission and d_instantiate to maintain handles.

	Simple syscall	Simple read	Simple write	Simple stat	Simple open/ close	Select 10 fd's	Select 500 fd's	Pipe latency	AF_UNIX latency	Process fork+ exit	Process fork+ /bin/sh -c	**Geometric mean**
unprotected	0.375	0.477	0.517	1.104	2.591	0.624	8.935	12.854	8.812	235.4	1830	
protected	0.376	0.526	0.580	1.122	5.867	0.624	8.958	13.994	9.785	249.8	1963	
Overhead (%)	0.09%	10.28%	12.15%	1.62%	126%	-0.1%	0.26%	8.87%	11.04%	6.08%	7.27%	**12%**

Figure 4: lmbench Performance overhead

When an object is associated with a subject (as a result of a file open, pipe or socket creation), the object will be attached to the subject via at least one handle. When the association is broken, e.g., due to a `close` operation, the corresponding handle is destroyed.

4.2 Constraint propagation

When a subject A opens a file O for writing (or a socket connection with another process), constraints from the file (or target process) have to be propagated in the inverse direction of information flow, as described in Section 3.4. The open operation is permitted if the invariants regarding `current_lbl` and `min_lbl` can be satisfied after this propagation.

Note that constraint propagation can involve circular dependencies as illustrated in Figure 2. To deal with cycles, we use a fixpoint algorithm for constraint propagation. To detect a fixpoint, our algorithm stores the previous value of `min_lbl` in a variable called `last_min_lbl`. It then updates the value of `min_lbl` of A to be the maximum of `last_min_lbl` and the label of the file O. If `min_lbl`$(A) =$ `last_min_lbl`(A) then a fixpoint has been reached, and our algorithm stops. If not, then the same process is used to propagate the new value of A's `min_lbl` to each of the subjects $S_1, \ldots, S_n$ that output to A, and the process continues. If any of the propagation steps fail because it results in a `min_lbl` exceeding the value of `current_lbl`, then the `open` operation is denied, and the values of `min_lbl` restored.

The same fixpoint algorithm is used even if A performs a `close` rather than an `open`. The only difference is that instead of computing the maximum of A's `min_lbl` and that of the new object being opened, we recompute `min_lbl` as the maximum of the labels of all the currently open write handles of A. However, in the presence of cycles, this simple algorithm will not always compute the least fixpoint. For this reason, our algorithm will retry constraint propagation from scratch before denying an open request. Note that (a) this retry step is unnecessary if no `close` operations have taken place since the last retry, and (b) constraint propagation itself is unnecessary for processes that are already at low integrity.

LSM provides no hooks on `close` operation: SRFD is not notified when a process closes a file. As a result, SRFD may have stale information regarding files opened. This requires SRFD to walk through the file descriptor table to prune out outdated handles when recomputing constraints. SRFD optimizes this by recomputing the constraints only when the current constraints cannot be satisfied.

4.3 Tracking subjects

Processes inherit a lot of rights from their parents, e.g., ability to write to a file. SRFD needs to be aware of these inherited rights to protect against self-revocation of these rights.

When a new process is created, SRFD duplicates the book-keeping information associated with the parent to the child. This approach automatically captures the communication between parent and child that happen using mechanisms such as pipes. The most common use of pipes occur in the context of shell processes, where the parent first creates a pipe with a readable-end and a writable-end. It then creates two child processes. At this point, the parent and children can all read and write from the pipes, so there is cyclic dependency between them. As a result, any constraint propagation will result in all three processes having the same `min_lbl`. However, in the next step, parent shell will close the two ends of the pipe, and then the first child will close the readable end of the pipe, while the second child will close the writable end of the pipe. After these close operations, there can be no flow between the children and the parent shell. Moreover, no information can flow from the second child to the first child. All of this is handled by our constraint propagation algorithm, which will correctly allow the second child to be downgraded (if necessary) without having to downgrade the first child or the parent.

4.4 Limitations

Our current prototype does not enforce its policies on operations relating to capabilities, file mount points, signals, message queue, and semaphores. In particular, low integrity processes performing these operations are not restricted. We also simply denies lower integrity processes to ptrace on higher integrity processes. We have left out these aspect since our experiments did not make use of these system calls. A complete implementation should also mediate these operations by propagating labels. It is part of our ongoing and future work to mediate them all.

For sockets, our prototype handles Unix domain sockets because the two ends of the socket connection are within the control of the OS. For sockets in the internet domain, their other end is typically outside the control of the OS. Hence SRFD does not attempt to enforce any policies on such internet sockets.

5. Evaluation

5.1 Performance

We evaluate the performance of our SRFD system using micro- as well as macro-benchmarks. All the evaluations are performed on a Ubuntu 13.10 VMware virtual machine allocated with one VCPU AMD Opteron Processor 4228 HE (2.8GHz) and 1GB RAM.

As a micro-benchmark, we use `lmbench`, which measures the overhead for making individual system calls. Figure 4 shows the overheads of our system for different classes of system calls. Note that the overheads are modest: the geometric mean is about 12%, and the arithmetic mean is 16%. Note that if we exclude open and close, which are typically less frequent than other calls such as read/write, the overheads are much smaller — less than 5%.

It is natural for `open` and `close` to have higher overheads because of constraint propagation, but that does not explain a doubling of execution time. It occurs in our prototype because LSM does not provide hooks for `close`, and as a result, our implementation has to walk through the list of open file descriptors while propagating constraints. In contrast, because there can be no failures on read and write, no additional checking is needed, and the only work is to blindly copy `current_lbl` from the source to destination.

Micro-benchmarks help to understand and explain the the overheads of kernel-based defenses such as ours, but they tend to overestimate the overheads because most applications spend only a minority of their time in the kernel. Macro-benchmarks are better at estimating overheads experienced by real users in practice. For this

	Unprotected Time (s)	Protected Overhead
400.perlbench	554.41	-0.21%
458.sjeng	865.29	-0.23%
462.libquantum	1032.35	-0.23%
471.omnetpp	543.24	0.27%
473.astar	738.29	0.16%
433.milc	875.47	-0.14%
Average		0.04%

Figure 5: SPEC2006 Overhead (showing top 6), `ref` input size

	Protected Overhead
Openssl	-0.08%
dpkg -b coreutils	2.93%
dpkg -b am-utils	1.22%
Firefox	4.89%
Postmark	5.74%

Figure 6: Overhead on other benchmarks

reason, we used several macro-benchmarks, including the CPU-intensive SPEC 2006 and openssl, file-system intensive `Postmark`, and commonly used programs such as browsers and software builds.

From Figures 5 and 6, it is clear that overheads on CPU-intensive programs such as SPEC and openssl are negligible — the overheads are below measurement errors/noise.

Package builds, which represent a combination of CPU and I/O load, show a slightly higher overhead of 1% to 3%. Specifically, our benchmark built Debian Linux packages for `coreutils` and `am-utils` from source code. Another mixed load consists of Firefox, whose overhead was measured using `pageloader`, a benchmarking tool from Mozilla. Top 3000 Alexa sites were prefetched in this experiment so as to eliminate the effects of network latency. (If this was not done, then the overheads will be even smaller.) The overhead experienced was 5%.

Finally, the I/O-intensive `Postmark` was configured to create 500 files with size between 500 bytes and 500 Kbytes. The overhead reported was 6%.

5.2 Experience

As noted in the introduction, our work is motivated by a continuing trend in sophisticated and adaptive malware attacks, and our desire to provide principled defenses against them. Existing approaches rely on techniques such as sandboxing a few key applications such as browsers and email readers that have the most exposure to malware. While sandboxing these applications can prevent some attacks, e.g., those that try to mount a code injection attack on an email reader (or other document viewers invoked by a browser), more sophisticated attacks can often get around these defenses. For instance, users may save a document on their desktop, and subsequently open it with their favorite document editor/viewer application. Since the application is typically not sandboxed in this usage scenario, the attack can succeed. In contrast, an information-flow based approach would mark such files as untrusted, and regardless of the number of applications that process them, or how many intermediate steps they go through, untrusted files will always be operated on by low integrity processes. Since such processes can only output low integrity files, and cannot modify high integrity files or interfere with high-integrity subjects, their attempts to compromise system integrity will continue to fail.

Although these theoretical benefits of information-flow based integrity protection are well-known, these techniques have not found widespread use on modern operating systems as they often pose compatibility challenges. In this section, we walk through several illustrative and common usage scenarios to demonstrate that SRFD can work well on contemporary operating system distributions, without posing major compatibility problems. Naturally, our focus will be on illustrating features specific to SRFD, as opposed to information-flow based techniques in general.

In these scenarios, we assume that the default OS installation consists of only high-integrity files; and that low integrity files enter the system when it begins to be used, and new files are created by untrusted subjects. We assume that browsers and email readers are run as low integrity processes.

5.2.1 Self-revocations involving files, pipelines and sockets

The scenarios discussed here illustrate the benefits of accurate information-flow dependency tracking in SRFD, and how that permits us to provide more functionality as compared to previous approaches (specifically, LOMAC [8]), while avoiding self-revocation.

One of the challenges in SRFD is to track communications between processes. This can be nontrivial when a deep pipeline is involved. Consider the command:

```
cat lowI | grep... | sed | ... | sort | uniq » highI
```

It is necessary to propagate labels across the pipeline to ensure that information from low-integrity file `lowI` is prevented from contaminating a high integrity file `highI`. Opportunities for self-revocation abound, especially if the shell opens `highI` before `cat` gets a chance to open `lowI`. Even otherwise, self-revocation is possible since intermediate commands such as `grep` may begin execution as high integrity processes, and then be prevented from reading their input pipes, or they may be downgraded and prevented from writing on their output pipes. LOMAC [8] avoids self-revocation on pipes by downgrading process groups at a time — in this case, all processes in the pipeline will be part of the same process group.

SRFD accurately captures information flow dependencies between the processes in the pipeline, and can avoid self-revocation, while preserving usability. In particular, depending on the order in which processes are scheduled, `cat` may be permitted to downgrade. In this case, SRFD will deny the open operation on `highI`. Alternatively, if `highI` is opened first, SRFD will deny `cat`'s attempt to open `lowI`.

Another example that illustrates the strength of SRFD is:

```
cat high1 | tee high2 | lowP
```

where `lowP` is a low integrity utility program. SRFD will run this pipeline successfully: both `cat` and `tee` will be remain at high integrity, and be able to output to high integrity file `high2`, while `lowP` will run at low integrity. LOMAC requires all processes in the pipeline to be at the same level, and hence cannot run this.

SRFD supports sockets, and can avoid self-revocation on processes that make use of these features. When a server program has a high integrity file opened for writing, SRFD will deny connections from a low integrity client, as the establishment of such a connection would violate the constraints on `min_lbl`. Moreover, any client that is already connected to such a server will be prevented from opening a low integrity file, or connecting to any other low-integrity process. LOMAC will experience self-revocation.

5.2.2 Commonly used applications

SRFD is implemented on a Ubuntu 13.10 desktop system. This system runs a large number of applications and servers, including a number of daemons, X-server, GNOME desktop environment, and so on. All these applications work with SRFD, but this is unsurprising: in our tests, these applications did not access low integrity files, and so SRFD does not constrain them in any way.

In the same manner, applications that don't modify high integrity files will run without any problems, as SRFD imposes no constraints on them. Most complex applications can be run this way — for instance, we run web browsers and email readers in this mode.

Most command-line programs can run as high or low integrity without any problems. Common utilities such as `tar`, `gzip`, `make`, compilers, and linkers can be run without any problems on low integrity files. Composing these command line applications using pipelines works as described in the preceding section. Thus, we focus the rest of this section on more complex GUI applications that need to access a combination of low and high integrity files.

Document viewers. Document viewers such as evince and Acrobat Reader can be used in SRFD without any issues. These programs can be used to open high and low integrity documents simultaneously. However, once the viewer has opened a low integrity file, it will not be able to overwrite a high integrity file.

Editors. GUI editors (e.g., gedit, OpenOffice, GIMP) impose additional challenges for dynamic downgrading systems like SRFD. When users select files to edit using file selection dialogs, applications tend to open every file to generate a preview, regardless of the integrity of the files. When users open a directory containing low integrity files, the editors will automatically be downgraded to low integrity even if the users did not intend to open low integrity files.

To prevent editors from downgraded accidentally, we can allow editors to be downgraded only when demanded by users. We can rely on the "implicit-explicit" mechanism suggested in [28] to identify file accesses that are requested explicitly by users, and only allow editors to be downgraded on opening these files. Other low integrity files can be denied when accessed implicitly.

Media Editors. We consider media editors (e.g., f-spot and audacity) separately because they usually do not modify the original media files directly. Instead, they edit copies of the media files. As a result, these media editors can be used without usability issues.

5.2.3 Defense against malware

We downloaded a rootkit `ark` from [3]. The tar file was labeled as low integrity when downloaded into the system by a web browser. The user then untars the file by invoking `tar`. SRFD started `tar` as a high integrity process, with `current_lbl` = Hi, `min_lbl` = Lo because it has no constraints on its output files and it has not been contaminated with any low integrity information. `tar` started by loading libraries like `ld.so.cache` and `libc − 2.17.so`. The `tar` process was then downgraded to low integrity when reading the rootkit tar file. `tar` process then spawned `gzip` as low integrity to decompress the file. After decompressing, the `tar` process continued to untar. All of the new files created are automatically labeled as low integrity.

With these integrity labels in place, SRFD can easily preserve system integrity. Specifically, system directories are labeled as high integrity and hence system utility rootkits cannot be placed in the system directories. However, it is possible for users to accidentally invoke these rootkits by placing them in some user-specific search paths. SRFD protects the system integrity by downgrading processes when these rootkits are executed or used, including executions by root processes. Hence, when a user process executes a low integrity binary or loads library, the process will be downgraded and is prevented from damaging system integrity.

SRFD also intercepts LSM hooks related to kernel modules. Low integrity kernel modules cannot be loaded even by root processes.

6. Related Work

A number of related works, including classical work on information flows and integrity protection were discussed in the introduction and the main body of the paper. We focus this section on related works that haven't been discussed before, and on providing a more in-depth comparison of works that are most closely related.

LOMAC [8] argues that a central reason for non-adoption of conventional information flow techniques is that of compatibility. They consider information flow systems that support privilege revision (such as dynamic downgrades) and those that don't, and conclude that former class provides increased compatibility.

They point out that policies such as low-watermark policy had not received much attention because of the self-revocation problem. They proceeded to address this problem in a particularly common case, namely, the pipelines created by shell processes. As noted earlier, their solution relied on the shell's use of UNIX process groups to run each pipeline, and ensuring that all processes within such a group had identical integrity labels. In this manner, there will never be a need to restrict communications within a process group, and thus self-revocation involving pipes is prevented. They remark that they "cannot entirely remove this pathological case without also removing the protective properties of the model." Indeed, the solution they present does not attempt to address revocations involving files, sockets, etc. Our work is inspired by their comments, and shows that it is in fact possible to retain the security benefits of integrity protection, as well as the compatibility benefits of privilege revision without incurring the cost of self-revocation.

Promoting early failure, as we do in this paper, is not the only way to solve the self-revocation problem. An alternative approach is to build recovery mechanisms to "roll back" failed executions. This is not always easy to do in general. One-way isolation [24] supports roll back as the default choice, while providing primitives to commit executions that the user determines to be secure. However, it is problematic to rely on users to decide what is secure. Not only does it demand considerable time, effort and skill on the part of users, but also suffers from the fact that users could be easily fooled. Thus, roll-back techniques coupled with automated procedures for determining secure executions are needed. Such automated procedures require full specification of what is secure — this itself is too difficult a task to be accomplished in general. However, it may be possible to specify detailed and accurate policies for secure execution in special cases. One example of this is the secure software installation [25] work, where a policy for determining secure installations was specified and checked automatically.

Roll-back based approaches complement our work. In particular, a complete SRFD system needs to support secure installation of untrusted code, and a technique such as SSI [25] can do this for us.

There has been a resurgence of interest in information-flow control in the last several years. Some of the techniques start off from classical *centralized* information-flow techniques, while others have targeted *decentralized* information flow control (DIFC). UMIP [14], IFEDAC [18], PPI [26] and PIP [28] belong to the first category. Another common thread among these approaches is that they target contemporary OSes, specifically, Linux. UMIP, IFEDAC and PPI all support dynamic downgrading of subject labels (i.e., LD). UMIP and IFEDAC do not address the problem of self-revocation, perhaps expecting that the applications will have mechanisms to deal with the problem. PPI relies on training for determining whether a certain access should be denied, or result in subject downgrade. Thus, it can eliminate some downgrades where training suggests that it will lead to failures. PIP uses early downgrading (ED) and hence does not suffer from self-revocation problem. However, as noted earlier, ED restricts functionality over LD

— thus, PIP avoids self-revocation at the cost of increased security failures. Furthermore, PIP relies on userid for policy enforcement, and hence cannot support low integrity root processes.

Some of the works that pursue the DIFC model include HiStar [30] and Asbestos [6], which redesign the operating systems to provide finer-granularity information flow control. Flume [13] provides DIFC within the context of standard OS abstractions. All of these works require nontrivial application or OS modifications in order to take advantage of information flow control. Changes to cope with self-revocations would be a small part of these modifications, and so self-revocation is not explicitly treated in these works.

Schneider [22] formulates enforceable security policies using the formalism of security automata. These automata make transitions that are entirely based on a subject's own operations, such as open's, read's and write's. Whereas these automata can only accept or reject an execution sequence, Ligatti et al [16] proposed a more powerful automata called *edit automata* that could also suppress or modify a subject's actions. We also use automata to compare different downgrading schemes for information flow systems, but the transitions in our automata are not only dependent on the subject's actions, but also the state of the file system. This is because whether an operation opens a high or low integrity file is a function of the file system state. Indeed, Ligatti et al [16] explicitly specify that security properties in their model are those that are purely functions of the operation sequence.

Policy-based confinement [5, 10, 11, 17, 21, 23] has been studied and widely deployed as a defense against malicious code. A runtime monitor allows or denies actions of processes based on a pre-defined policy. Depending on the enforcement mechanism used, the implementation can be tricky [9, 12] due to TOCTTOU attacks. The most difficult part of applying these techniques is to have a good policy to identify bad behaviors [19]. A policy that is too permissive would let malicious programs to compromise the system, while a policy that is restrictive would impair usability.

Isolation [15, 25, 24, 20] is another commonly used technique to protect system integrity. By running potentially malicious code in an isolated environment, the host system integrity can be preserved. It is simpler than policy-based confinement because there is no application-specific policy required. All resources are isolated. A main drawback of isolation is fragmentation of the file system namespace into several distinct "isolated" namespaces. When a user wants to access a file, they first need to recall which container has this file. Moreover, if they want to combine information across multiple containers, it is not only cumbersome, but opens an avenue for malicious code or data in one container to infect another.

7. Conclusion

We categorized information flow policies into No Downgrading (ND), Eager Downgrading (ED) and Lazy Downgrading (LD). We proposed a formal model to compare these information flow policies in terms of functionality and compatibility. Our model shows that LD is more functional than ED, which, in turn, is more functional than ND. However, LD poses compatibility problems due to self-revocation, whereas ND and ED do not suffer from this drawback. We therefore proposed SRFD, which combines LD's functionality with the compatibility of ED. We formally showed that SRFD does not suffer from self-revocation. We also showed that unless an oracle was available to predict future behaviors of programs, it is not possible to further improve SRFD, i.e., accept more executions without compromising integrity or risking self-revocation. Our prototype shows that SRFD provides very good performance, and can support a variety of benign usage scenarios while providing principled defense against malware attacks. We believe that this work represents a promising step that can contribute to some mainstream adoption of information-flow based integrity protection techniques. To further this cause, we are releasing the source code for our prototype [27].

8. References

[1] Akari, `http://akari.sourceforge.jp/`.

[2] Operation Aurora, `http://en.wikipedia.org/wiki/Operation_Aurora`.

[3] Packet Storm, `http://packetstormsecurity.com`.

[4] K. J. Biba. Integrity Considerations for Secure Computer Systems. In *Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts*, 1977.

[5] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *LISA*, 2000.

[6] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. In *SOSP*, 2005.

[7] N. Falliere, L. Murchu, and E. Chien. W32. Stuxnet Dossier. *White paper, Symantec Corp., Security Response*, 2011.

[8] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *S&P*, 2000.

[9] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *NDSS*, 2003.

[10] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*, 2004.

[11] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In *USENIX Security*, 1996.

[12] K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *NDSS*, 2000.

[13] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *SOSP*, 2007.

[14] N. Li, Z. Mao, and H. Chen. Usable Mandatory Integrity Protection for Operating Systems . In *S&P*, 2007.

[15] Z. Liang, W. Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An Isolated Environment for Experimenting with Untrusted Software. In *TISSEC 12(3)*, 2009.

[16] J. Ligatti, L. Bauer, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-Time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[17] P. Loscocco and S. Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Ottawa Linux symposium*, 2001.

[18] Z. Mao, N. Li, H. Chen, and X. Jiang. Combining Discretionary Policy with Mandatory Information Flow in Operating Systems. In *TISSEC 14(3)*, 2011.

[19] C. Parampalli, R. Sekar, and R. Johnson. A Practical Mimicry Attack Against Powerful System-Call Monitors. In *ASIACCS*, 2008.

[20] S. Potter and J. Nieh. Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems. In *USENIX conference on USENIX annual technical conference*, 2010.

[21] N. Provos. Improving Host Security with System Call Policies. In *USENIX Security*, 2003.

[22] F. B. Schneider. Enforceable Security Policies. In *TISSEC 3(1)*, 2000.

[23] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. In *SOSP*, 2003.

[24] W. Sun, Z. Liang, V. N. Venkatakrishnan, and R. Sekar. One-Way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *NDSS*, 2005.

[25] W. Sun, R. Sekar, Z. Liang, and V. N. Venkatakrishnan. Expanding Malware Defense by Securing Software Installations. In *DIMVA*, 2008.

[26] W. Sun, R. Sekar, G. Poothia, and T. Karandikar. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *S&P*, 2008.

[27] W. K. Sze and B. Mital. Self-Revocation Free Downgrading (SRFD). `http://www.seclab.cs.sunysb.edu/seclab/srfd`.

[28] W. K. Sze and R. Sekar. A Portable User-Level Approach for System-wide Integrity Protection. In *ACSAC*, 2013.

[29] TrendLabs APT Research Team. Spear-Phishing Email: Most Favored APT Attack Bait. 2012.

[30] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *OSDI*, 2006.