Stony Brook University

A Framework for Enforcing Information Flow Policies

A Thesis Presented
by

Bhuvan Mital

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

May 2010

Stony Brook University

The Graduate School

Bhuvan Mital

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Dr. R. C. Sekar, Thesis Advisor
Professor, Computer Science

Dr. Scott Stoller, Thesis Committee Chair
Professor, Computer Science

Dr. Robert Johnson
Assistant Professor, Computer Science

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

A Framework for Enforcing Information Flow Policies

by

Bhuvan Mital

Master of Science

in

Computer Science

Stony Brook University

2010

Reactive approaches for ensuring security, like signature based scanning and behavior monitoring, have been around for quite some time. However they have failed to provide assurances about overall system integrity, and can easily be defeated by sophisticated techniques like code obfuscation and encryption. Another class of attacks includes those that occur in multiple steps (often referred to as *multi-step* attacks). Information flow based approaches provide a basis for mediating and tracking dependencies between system entities, and can thus prove to be helpful in overcoming these shortcomings. However, success in applying information flow based techniques to modern COTS operating systems has been limited, since a strict application of information flow policy can break existing applications and OS services. One common case of poor usability is when an application is denied write access to a high integrity file in the middle of the write-operation as a result of reading from a low integrity file.

Our framework attempts to address this issue of loss in usability by maintaining integrity constraints for each subject (process) and object (files, sockets, IPC channels etc.) in the system, and permitting or denying access requests by ensuring that no invariant is violated. To achieve this, our approach maintains a per-process list of objects being accessed. For each new read-open request made by an application, our policy enforcer propagates integrity constraints from the objects in the application's list to the new object that the application wishes to open. The success or failure of the request then depends on the new object's ability to honor these constraints. This strategy restricts service denials to early failures, which the applications handle far more gracefully than read or write denials. To provide completeness to the solution, our framework enforces policies for all different types of objects (files, links, pipes, sockets, devices, IPC channels). The implementation of our framework utilizes Linux Security Module (LSM) hooks. A considerable portion of our work also deals with understanding and documenting the flow of the Linux kernel code involved in the LSM framework and mapping the abstract operations of our framework to the appropriate LSM hooks.

To my parents, Priyam, Kanishk and Mala.

Contents

List of Figures

List of Tables

Acknowledgments

My immense gratitude to my advisor, Dr. R.C. Sekar, for his constant guidance and motivation. I thank Dr. Scott Stoller and Dr. Robert Johnson for being on my defense committee, and providing valuable suggestions. I am thankful to everyone at the Secure Systems Lab (SSL), for making it a great learning experience. I also want to specially thank Prachi Deshmukh, for her formidable support and help through this project.

This thesis was made possible in part thanks to the ONR grant 000140710928. The thesis is also sponsored in part by the NSF grant CNS-0831298.

Chapter 1

Introduction

Security threats and malware attacks have increased dramatically over the past few years. With Internet becoming powerful, its user-base has increased massively and so has the threat from malware, which gets downloaded to the victim's machine either actively or passively. Malware defenses that use reactive approaches, such as signature-based scanning and behavior monitoring, can be defeated by code encryption and obfuscation. Sophisticated attackers can easily evade detection by such approaches by simply changing the structure and behavior of their malware.

Sandboxing based security solutions restrict the number of resources that an application can access and modify. Many proactive defenses based on similar ideas, are used commonly against untrusted software, to limit the set of system resources that can be modified by potentially malicious processes and restrict communication with other system processes. However, many such techniques do not mediate read-accesses and hence there is no enforcement for data that is read from low integrity sources, such as malware outputs. In the absence of read-access mediation, *multi-step* attacks can still be perpetrated, as simply as malware using vi-editor to write to a high integrity file, which it otherwise is not authorized to write to, directly. An example of a real world attack is that of Windows Vista Security Model which does not have security enforcement policies for *read-downs* (reads that occur from low-integrity sources) and only enforces policies that mediate and thus prevent *write-ups* (writes that occur on high-integrity objects, whose integrity must be preserved). The attacker simply overwrites the Vista start-menu file, an action that does not involve escalated privilege level. The affected start-menu items now points to entry-points in the malware rather than the usual applications. However, the naive user is oblivious of this and trusts the start-menu entries. Thus when he clicks on a menu item and is prompted by the Vista UAC, which requests higher privilege level for running the intended program, the user almost certainly allows the escalation of privileges. The malware then executes with escalated privileges thereby compromising system security. Thus due to the absence of read access policy enforcement in the Windows Vista Security Model, it becomes vulnerable to a potential multi-step attack.

It may be worth noticing that while read accesses refer to both *read-ups* and *read-downs*, from an integrity standpoint, *read-ups* are never an issue. Like *read-ups*, *write-downs* too are never problematic in the integrity model. Hence any reference to an integrity-preserving security solution's inability to mediate read accesses, essentially means it's inability to mediate read-downs.

The same holds true with regards write accesses.

Information-flow based integrity preservation techniques, which regulate write-access as well as read-access, and mediate flow of information (occurring on all possible channels of communication) between the dependent entities in a system, can be used to counter multi-step attacks. An early, and significant work on information-flow based techniques is the Biba integrity model [4] which serves as a guiding principle for similar works. However Biba model is very strict and simply denies all *write-ups* and *read-downs*, thus it tends to break the functionality of many applications and suffers from poor usability. *Low Watermark* model [5, 11] builds over the Biba model by adding a "subject downgrade" policy. This means that a subject gets downgraded to a lower integrity level when it reads from a low integrity object, and subsequently runs at a reduced integrity level for the remainder of its lifespan. The *downgrade* policy, addresses some of the usability issues. However it does not address the *self-revocation* problem where an application is denied write access to a high integrity file in the middle of the write-operation, consequent to getting downgraded upon reading from a low integrity source. An example is that of a peer-to-peer downloader that downloads data from high integrity sources/sockets, gets downgraded in the middle of this operation if the user tries to download from a low integrity socket, and then can't complete its prior task, which involves writing to high integrity files. Another example is when a user edits a high integrity file in a text editor application, and in the middle of operation, the user opens a low integrity file for reading. Now this operation causes the test editor to get downgraded and it thus loses it's write ability on the high integrity file. The user may either lose important unsaved modifications to the high integrity file or in the worst case it may leave the file in an inconsistent state.

PPI [14] uses the concept of *trusted applications* which means applications which are designated as trusted, do not get downgraded to lower integrity levels even when they consume low integrity inputs. This concept can be applied in solving the above mentioned problem of ensuring usability, by simply making the peer-to-peer downloader and the text editor, trusted. However trust alone cannot be the solution to this problem, because unlimited trust can become an undesired feature of a security model.

Our work derives motivation from this problem of loss in usability of applications, which arises due to enforcement of information-flow based policies such as *deny* and *downgrade*, while keeping *trust* to a minimum. In this thesis, we develop an integrity preservation framework for PPI. The high level goals of our framework are enlisted below and are discussed in detail in Section 1.2.

- *Promotion of Early Failures by Propagating Integrity Constraints*. Our focus is on promoting early failures as opposed to delayed failures, while preserving system integrity. Without loss of generality we can claim that early failures cause much less breakage in usability, than delayed failures. One example of this is that applications handle file-open errors far more gracefully than read/write errors. Another example is that of sockets: denying connection to a socket is better than denying send/receive on an established connection, at a later stage.
- *Completeness of Approach*. Our framework enforces integrity policies on all types of objects in the system. These include regular files, directories, sockets, pipes, links and various IPC communication channels.
- *Limiting Trust*. Our framework uses PPI's concept of trust and exempts trusted applications

from information flow policies. However the framework provides the flexibility, to either limit a subject's trust to a certain threshold or make the subject untrusted altogether. This helps our framework in limiting trust to only a few applications, e.g. ssh server is trusted for all input on port 22.

- *Flexible and Scalable Labels*. Each subject and object (named as well as un-named), in our system, has an associated integrity label. Our framework implements a label as an abstract datatype, with its domain being a lattice. Labels may also be used to encode both confidentiality and integrity at the same time. In the simplest case, we could still use a linear lattice, with the top element corresponding to highest integrity and lowest confidentiality; and the bottom element corresponding to lowest integrity and highest confidentiality. The current framework is designed to support all the suggested alternatives, for label implementation.
- *Fitting the framework in a contemporary Operating System*. The greatest challenge we faced was mapping our framework operations in a contemporary operating system. Linux operating System provides the LSM (*Linux Security Module*) framework for writing loadable security modules. We faced multiple engineering issues while mapping the abstract operations of our framework with the corresponding kernel hooks of the LSM framework, many of which stem from LSM's limitation. However the robustness of our design permits the seemless integration of our framework with any operating system that offers a security framework similar to LSM.

Our results indicate that our framework preserves integrity and maintains usability at the same time. The overall system overhead is marginal and the performance penalty does not change drastically with the increase in the number of active processes in the system.

The rest of the thesis is organized as follows. In Chapter 2 we talk about the details of the design of the PPI-Framework. We describe the details of our implementation and details pertaining to mapping with the LSM framework in Chapter 3. We present the actual experimental results obtained from the various benchmarks in Chapter 4. Chapter 5 provides references to some related works, and we conclude in Chapter 6.

Our framework is based on the PPI [14] approach for preserving system integrity. Section 1.1 gives an overview of the approach developed by PPI [14] for integrity preservation, presents the concept of trust, introduced by PPI and mentions how our framework implements this concept. In Section 1.2, we discuss the contributions of our framework.

1.1 Overview of the PPI Approach

Figure 1.1 illustrates the integrity and trust levels used in our framework, as advocated by PPI (Practical Proactive Integrity Preservation). To simplify the illustration, we use just two integrity levels: *high* and *low*. A subset of high-integrity objects are identified as *integrity-critical*. Integrity critical objects are those, whose integrity must be preserved under all circumstances. In other words, they are those objects that must never get downgraded. Integrity critical objects provide the basis for defining system integrity:

System Integrity: *System integrity is preserved as long as all integrity-critical objects have high integrity labels.* [14]

The initial set of initial integrity-critical objects is externally specified by a system administra-

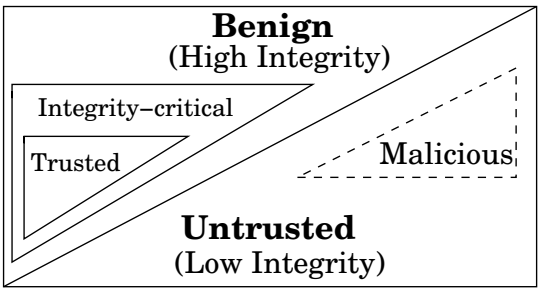

Figure 1.1: Classification of Applications in PPI

tor or the user of the system, both of who are assisted by automated analysis techniques specified by PPI. Files corresponding to applications that come from the OS vendor or are downloaded from trusted repositories have high integrities. Such applications are termed as *benign*. Such applications retain their trust labels as long as they do not consume data from low integrity channels (files, sockets etc.). Otherwise they get downgraded to lower integrity level. As an example, a bit torrent client application may get downgraded upon reading data from a low-integrity socket.

However it may be necessary to have a certain amount of trust on certain applications (e.g., certain webservers), which sanitize lower integrity data before consuming it. Trusting certain applications, may be absolutely necessary for their correct functioning, e.g. an ssh server must be trusted for all inputs received on port 22. Such applications are termed as *trusted*.

Our framework implements this concept of trust by making the applications *invulnerable* to low-integrity inputs. Such applications do not get downgraded as long as they can *sanitize* their inputs. The extent to which such applications can be trusted is defined by their *invulnerability level* which simply means that the application is invulnerable till its invulnerability level. Any input it consumes below this level will lead to its downgrading. Details on how our framework implements this feature is described in Section 2.1.6.

PPI develops a new approach for proactive integrity protection by overcoming the issues that information flow based systems suffer from, some of which were discussed earlier. PPI assigns *labels* to all the entities (objects and subjects) in a system and decouples these integrity labels from *access policies*. An integrity label on an object simply indicates whether its content is trustworthy and does not dictate whether trustworthiness must be preserved. On the other hand, a policy, is an indicator of whether an access (read or write) should be allowed or denied.

PPI offers the following policies when a high-integrity subject attempts to read a low-integrity object:

- *Deny* : deny the access
- *Downgrade* : downgrade the label of the subject to low-integrity and allow the operation
- *Trust* : trust the subject to protect itself without downgrading the subject.

PPI also offers the following options when a low-integrity subject attempts to write a high-integrity file:

- *Deny* : deny the access
- *Downgrade* : downgrade the label of the object to low-integrity and allow the operation

PPI also develops an analysis for automating the generation of integrity labels and policies that aim towards preserving the usability of applications in most cases. Details of this analysis and the related approaches have been described in the PPI [14] paper.

1.2 Contributions of the Framework

This section summarizes the contributions of our work. These primarily enumerate the unique features of our framework and some significant research that we did while implementing this framework.

Promotion of Early Failures by Propagating Integrity Constraints Our framework promotes early failures as opposed to delayed failures, as already mentioned in Chapter 1. In order to achieve this, our framework maintains certain integrity invariants (as per the integrity policies). The policy checks and invariant maintenance is completely decoupled from the integrity labels associated each object and subject in the system. These label values are assigned/read, validated and modified (if necessary) at runtime as per the integrity policy and the access is permitted only if the pre-defined integrity invariants are satisfied. As an example, when a subject S_1 requests access to an object O_1 in read-mode, the framework propagates the constraints, from all objects that S_1 is *already* accessing in write-mode, to O_1. The access is granted only if O_1 can satisfy the constraints. Likewise, when a subject S_1 requests access to an object O_1 in write-mode, the framework propagates the constraints, from all subjects that are *already* accessing O_1 in read-mode, to S_1. The access is granted only if S_1 can satisfy the constraints. It is easy to observe that the constraints are propagated, in the direction opposite to that of information flow. Our technique clearly ensures that communication establishment with a new channels is permitted only if it does not lead to a an access denial on an existing channel of communication. Thus our framework also addresses the problem of *self-revocation*, because service denials (if any) are limited to early open-request failures.

Flexible and Scalable Labels Each subject and object (named as well as un-named), has an associated label. Our framework implements a label as an abstract datatype, with its domain being a lattice. PPI [14], envisioned label as a linear lattice with 8 levels, with level 7 and level 0 corresponding respectively to the highest and lowest possible integrity levels. But with our framework, the labels can be easily changed, say, to permit 1024 levels of integrity. Another obvious generalization is to support partial orders on labels rather than just total orders. This may be useful for handling information from different sources that we do not trust fully, but at the same time, we want to distinguish between these sources in order to ensure that one of these sources does not have the ability to compromise information from other sources. One possible way would be to break up each integrity level into many incomparable families, one corresponding to each such source, by adding another component to the label that specifies the source. Labels may also be used to encode both confidentiality and integrity at the same time. In the simplest case, we could still use a linear lattice, with the top element corresponding to highest integrity and lowest confidentiality; and the bottom element corresponding to lowest integrity and highest confidentiality. However, it would be more convenient to represent confidentiality and integrity components separately, i.e.,

let each label be a pair $\langle l_i, l_c \rangle$, where l_i specifies integrity and l_c specifies confidentiality. It is noteworthy that in such a label, the ordering on integrity and confidentiality go in opposite directions. In particular, a label $l_1 \geq l_2$ means that the integrity component of l_1's label is greater than or equal to that of l_2, while its confidentiality component is less than or equal to that of l_2. Our framework has been designed to supports all the suggested alternatives, for label implementation.

Completeness of Approach Our framework mediates all accesses by a process (*subject*) on various *objects* such as regular files, directories, sockets, pipes, links and various IPC communication channels. By enforcing policies on all types of objects in the system, our framework ensures completeness.

Limiting Trust Unrestricted trust can often turn out to be an undesired feature of a sound security model. Our framework gives the provision to restrict the invulnerability that a subject can exercise, when reading from low-integrity inputs, by defining an invulnerability level for each subject. In the most common case, the invulnerability of a subject could simply be turned off, which means it is no longer a trusted subject.

Mapping the Framework Abstract Operations to the LSM Hooks Our framework achieves its objectives by defining a set of abstract operations that are discussed in detail in Section 2.2.2. Fitting the framework in a contemporary OS presents multiple challenges and engineering issues. We implemented our framework for the Linux operating system by mapping our abstract operations to the hooks in the Linux Security Module (LSM) Framework. This required careful study of the hooks and understanding the control flow between the different hooks. The success of our framework validates our study and understanding of the LSM hooks. Our flow graphs (referred in Appendix-A) serve as a good starting point for someone who wishes to understand the LSM framework.

Chapter 2

Framework Design Details

This chapter discusses the design details and the design decisions that we made while developing an enforcement framework for PPI.

2.1 Approach Overview

Figure 2.1 provides a high-level illustration of information flows in our framework. Policy enforcement is effected using LSM hooks in the kernel, and the details of this enforcement will be described in the subsequent sections.

2.1.1 Labels

An information label is associated with each entity on the operating system that can serve as a source, sink or conduit of information. A label is an abstract datatype, with its domain being a lattice. Initially, in PPI, we envisioned a linear lattice with 8 levels, with level 7 and level 0 corresponding respectively to the highest and lowest possible integrity levels. But this can be easily changed, say, to permit 1024 levels of integrity. Another obvious generalization is to support partial orders on labels rather than just total orders. This may be useful for handling information from different sources that we do not trust fully, but at the same time, we want to distinguish between these sources in order to ensure that one of these sources does not have the ability to compromise information from other sources. One possible way would be to break up each integrity level into many incomparable families, one corresponding to each such source, by adding another component to the label that specifies the source. The current framework is intended to support all these alternatives.

2.1.2 Objects, Subjects and Handles

The entities involved in information flow are the following:

Objects: They 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. 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). Best example of a pipe-like object is a socket.

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

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

Handles also provide a mechanism for a subject to distinguish between different objects with which it communicates. In particular, our framework provides mechanisms for a subject to customize information flow policies on a per-object basis by setting label attributes on the handle used (by the subject) to access that object. As an example, a subject may want to exercise invulnerability on a specific input and not on another.

Handles are conceptually similar to a 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.

2.1.3 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. To ensure this, our framework may dynamically downgrade the label of the destination. To prevent undesirable downgrading, a minimum label (called `min_lbl`) is associated with every subject and object. In the context of an earlier example on text editors (Chapter 1), the self revocation problem can be solved by ensuring that the `min_lbl` of the text editor remains greater than or equal to the `min_lbl` of the high-integrity file that it was writing to before it read low-integrity data. In other words this ensures that the editor never downgrades to an integrity level lower than the lowest integrity level of the high-integrity file.

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.

Note that if the label incorporates both integrity and confidentiality components, then the `min_lbl` of a destination will specify the lowest possible integrity and highest possible confidentiality of data that flows into that destination.

Finer-granularity control over downgrading is provided for subjects using a discretionary minimum label (`discr_min_lbl`). Note that a subject may get downgraded when it performs a read-open of an object with a lower label, or if one of its currently open (for reading) objects gets downgraded. In either case, a subject S may want to control its downgrades, i.e., it may be willing

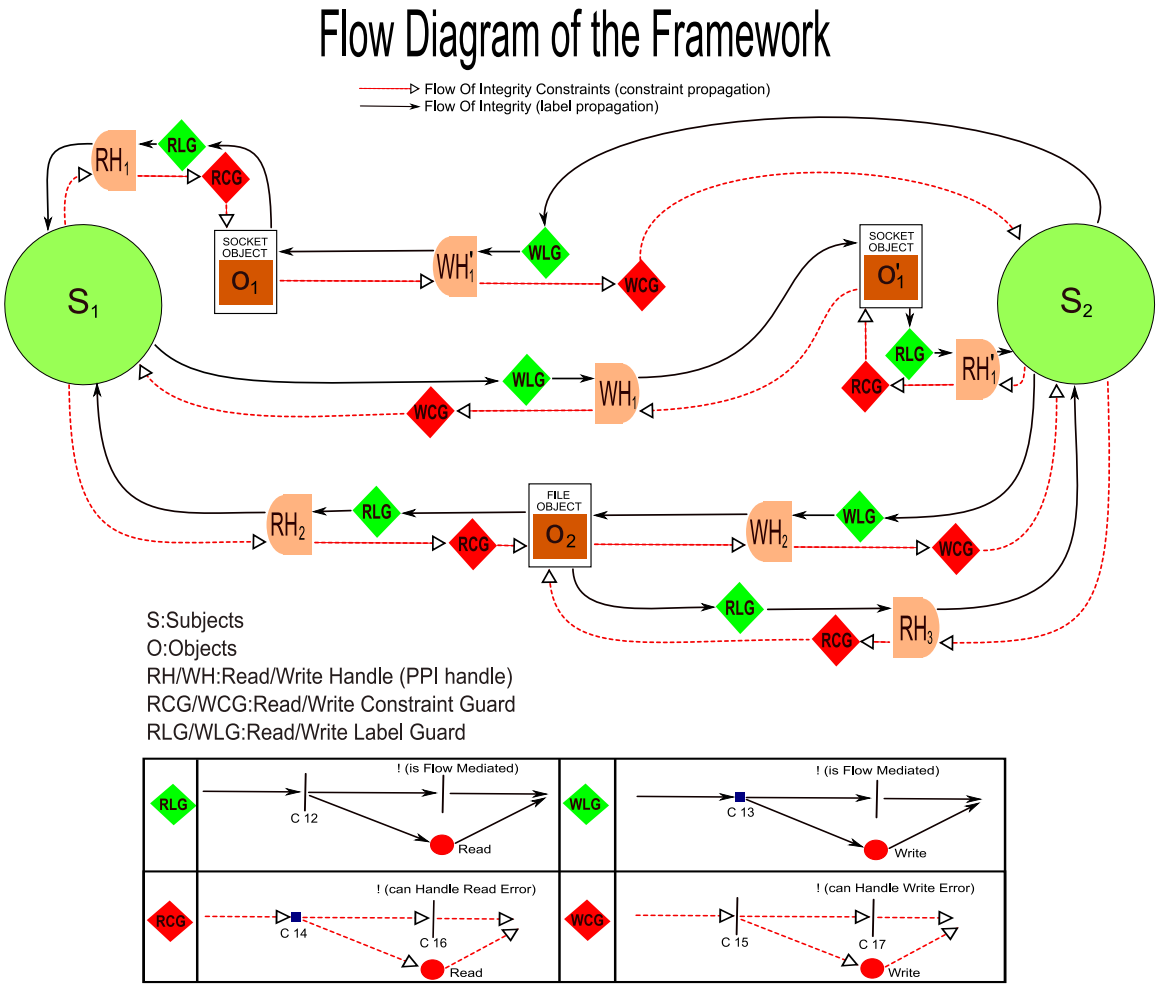

Figure 2.1: Illustration of Information Flow in our Framework
C 12, C 13, C 14 and C 15 indicate integrity constraints discussed in the Section A.1.4

to be downgraded to a level l_1 when reading from an object O_1, and another level l_2 when reading from another object O_2. S can achieve this by setting `discr_min_lbl` to l_1 on the read handle associated with O_1, and to l_2 for the read handle associated with O_2.

2.1.4 Forward information flows

Figure 2.1 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 receives the label of the subject writing to it. 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 S_1

subsequently reads from the object O_2, the label of O_2 will flow into S_1. This implies that for file-like objects, their read handles simply inherit the label of the corresponding objects. The boxes labeled RLG ("read-label guard") and WLG ("write-label guard") capture the fact that an actual flow between an object and a subject does not take place until the next read or write operation, except in cases where the file object is memory-mapped. (Another reason preventing the flow is if the subject is invulnerable, as described in Section 2.1.6.)

Since a socket is a pipe-like object representing two distinct flows, we split it into two peer objects O_1, the socket held by S_1, and O_1', the socket held by S_2. S_1 uses a read-handle RH_1 and a write-handle WH_1 to read from and write into the socket, while S_2 uses RH_1' and WH_1' respectively for the same purpose. Objects O_1 and O_1' are peers in the sense that the data written into one of these objects is read via the other. Data written by S_1 into the socket using the handle WH_1 is shown as flowing into the object O_1'. This way, when S_2 reads the socket using the handle RH_1', the label returned to S_2 will be that of the data written using WH_1. This explains why S_1's write-handle is associated with its peer's socket object O_1', while its read-handle is associated with its own socket object. Similarly, S_2's write-handle is associated with its peer's socket object, while its read-handle is associated with its own socket object. With these associations, we can once again say that handles inherit the labels of their associated objects, with read-label and write-label guards playing the same role as before.

2.1.5 Promoting early security failures via reverse constraint flows

Figure 2.1 also shows flows taking place via dashed arrows in the reverse direction of normal information flow. The purpose of these flows is to promote early rather than delayed security failures. An open failure, a special case of early failure, which occurs when a subject opens an object (like a file), is much better handled by most applications than delayed failure, which can happen on any read or write operation. This is because most applications are written to anticipate security violations on open operations, but not on reads or writes. This is also a key feature in our framework design.

Early failures are promoted by interpreting local security policies on entities (i.e., policies associated with individual objects and subjects) as constraints on them, and propagating these constraints across communicating entities via the dashed arrows. In particular, we consider constraints on minimum security labels (called `min_lbl`) that an object or subject needs to maintain, and propagate these constraints "upstream" to entities that produce the information flowing into these subjects or objects. The idea is that a downstream entity cannot satisfy a constraint that cannot be ensured by an upstream entity that produces the information flowing into the downstream entity. We associate a label called `current_min_lbl` on each entity to denote the minimum label derived for that entity using this constraint propagation process.

Whereas the forward flow of labels is normally delayed until an explicit read or write operation, constraint propagation, by default, is instantaneous. Since the whole purpose of constraint propagation is to avoid security failures on read/write operations, it would make no sense to delay propagation. However, subjects could indicate that they are capable of handling read or write errors on specific channels, and for those channels, constraint propagation is delayed. This is captured by the boxes labeled RCG ("read constraint guard") and WCG ("write constraint guard").

With the above constraint propagation in place, our framework will not need to return security

errors on read or write operations, but instead, such errors will mostly be confined to open operations. (In contrast, downgrades may happen on any read.) In effect, this requires our approach to be conservative: if a read or write security violation is possible some time in the future after an open, then that open should be denied.

Our framework uses `current_min_lbl` to identify possible future violations. When a subject S attempts to open an object O for reading, this open is denied if the object's `current_min_lbl` is less than the `current_min_lbl` of the subject S and the object's `current_min_lbl` cannot be increased to the level of the subject's `current_min_lbl`. Note that this happens even if `current_lbl`$(O) \geq$ `current_lbl`(S), i.e., the open operation would not immediately violate the information-flow policy. An application can be selective in terms of which inputs it is willing to deal with delayed failures (i.e., cope with read errors), and this can be done by setting `can_handle_errors` on specific handles. (At the time of open, a read handle inherits the subject's value of `can_handle_read_errors`.)

Write errors are worse than read errors: for instance, in the case of output files, denying a write requires that file to be closed midway, potentially leaving the file contents in an inconsistent state. So, our framework attempts to prevent write security violations. In particular, it is safe to assume that the output of a subject S will never go below its `current_min_lbl`. Our framework checks at the time of a write-open on object O if `current_min_lbl`$(S) \geq$`current_min_lbl`(O). If not, the open will be denied if subject's `current_min_lbl` can not be increased to the object's `current_min_lbl`. As in the case of reads, an application can choose to accept delayed failures on writes by setting `can_handle_write_errors` on specific handles. However, care must be exercised here, as attacks can be perpetrated on benign programs by compromised inputs: in particular, if the input has a low label, a subject may get downgraded, causing an already open output file to be truncated. To safeguard against this, a subject is permitted to set a write handle's `can_handle_errors` only if the subject's `can_handle_write_errors` is set. Similarly, it may set a read handle's `can_handle_errors` only if its own `can_handle_read_errors` is set.

2.1.6 Data validation and sanitization

Strict enforcement of information flow policies can break some applications. To mitigate this problem, PPI, like other information flow based techniques, allows subjects to be designated as *trusted*, and these subjects are exempted from policies. Rather than providing a mechanism that allows trusted subjects to indiscriminately violate all information flow policies, our framework provides two primitives that are narrower in scope. In particular, our framework provides mechanisms to model the fact that certain subjects perform adequate validation or sanitization of inputs or outputs on certain channels.

An untrusted subject is not permitted to read data with a label that is lower than its own. A trusted subject, on the other hand, can be permitted to read data with lower labels without decreasing its own label. Our mechanism for supporting this is designed to accommodate applications that perform such validation/sanitization selectively on certain inputs. This is achieved by associating an *invulnerability level* on the handles. A read operation will be permitted using a handle as long as the label of the data read is greater than or equal to the invulnerability level of that handle.

Note that input invulnerability models the ability of a trusted subject to perform adequate

input validation (or sanitization of confidential data). To express the fact that adequate validation/sanitization is performed on some inputs and not others, subjects can associate different invulnerability labels on different handles. Newly created read (write) handles inherit the input (output) invulnerability level associated with the subject.

Although our focus in PPI is on integrity, invulnerabilities can be used to handle trusted subjects that handle confidential data. A subject performing input sanitization (declassification) can set the confidentiality component of its invulnerability to be higher than its own, which means that it is able to read data that is more confidential than what is permitted by the subject's confidentiality label. However, for most programs, it is preferable to do such sanitizations at the output stage: one can state with a lot more confidence that a certain output is free of confidential information, than to say that confidential data is "scrubbed" right at the input. Indeed, many programs store confidential data in their memory, and some operations (e.g., dumping of core) will result in files containing this confidential data. The label on this file will not reflect this confidential content if we rely on input invulnerabilities. On the other hand, it is possible that some subjects are able to selectively read public components of files that contain a combination of confidential and public data *without ever needing to read confidential data into their address space*. Such subjects may be modeled using input invulnerability.

Analogous to input invulnerability, we can define a notion of *output invulnerabilities* that enable a (trusted) subject to output data that has a higher label than its own. Once again, this should be done selectively on those output handles for which the subject performs adequate validation and/or sanitization. This kind of output validation is analogous to an *endorsement*. Specifically, the data output using a handle is given the maximum of the labels of output invulnerability of that handle, and the current label of the handle. Note that it makes much more sense for trusted subjects to be permitted to output data at a lower confidentiality level than their own, e.g., a server handling sensitive data that ensures that an output returned to an untrusted user is free of sensitive content. The case for using output invulnerabilities to increase output integrity is much weaker. However our framework is capable of incorporating a confidentiality model, as a part of future work, so the notion of output invulnerabilities has been retained, without being used in the present context of integrity.

Finally, we address the question of trust, and how it is specified. In our framework, trust is specified using the subject's `invul_lbl`: it can exercise its input or output invulnerabilities when `current_lbl > invul_lbl`. (For untrusted subjects, `invul_lbl` defaults to the value of `current_lbl`.) In addition, two additional labels are associated with subjects: `input_invul` and `output_invul`, which specify the default invulnerability labels on newly created input and output handles respectively.

2.1.7 Discretionary policies

Our framework permits subjects to impose stringer constraints on their labels than the one implied by existing policies on the subject and the objects/subjects that it communicates with. Specifically, a subject can set its `discr_min_lbl` to be higher than its `min_lbl`. The constraint propagation mechanism described earlier will ensure that the label of the subject remains above `discr_min_lbl`. Similarly, we introduce `discr_invul`, `discr_input_invul` and `discr_output_invul`.

Note that discretionary labels cannot be used to violate mandatory policies. For instance, `discr_min_lbl` can't be set to a value less than `min_lbl`; `discr_input_invul` can't be set to a value less than `input_invul`; `discr_output_invul` can't be set to a value more than `output_invul`.

Newly created input handles inherit the value of the subject's `discr_input_invul`, while newly created write-handles inherit the value of the subject's `discr_output_invul`. Thus, the main purpose of a subject's `discr_input_invul` and `discr_output_invul` is to provide a mechanism for the subject to control the initial value of `discr_input_invul` and `discr_output_invul` for newly opened handles. An invulnerable subject *cannot* lose its invulnerability till all its read handles exercising invulnerability are closed. For achieving this, we deny all attempts to open new read handles,by a subject, that may potentially reduce the `current_lbl` of the subject to the extent that it loses its invulnerability.

In addition, a subject can express its ability to handle security violations on read (or write) operations using a flag `can_handle_read_errors` (`can_handle_write_errors`). These flags are inherited by newly created handles. In addition, these flags can be modified on individual handles by the subject, provided they are consistent with subject's ability to handle read or write errors.

2.1.8 Enforcement

It is important to note that permit/deny decisions are largely orthogonal to constraint propagation. In particular, basic policy enforcement (i.e., permit/deny decisions) are made purely on the basis of local information: specifically, `current_lbl` and `current_min_lbl` on the entities involved in an interaction. When a flow is about to occur from a source A to destination B, our framework will check if

$$\texttt{min\_lbl}(B) \leq \texttt{current\_min\_lbl}(B) \leq \texttt{current\_lbl}(A)$$

If so, B's `current_lbl` will be set to that of A, and the operation permitted. Otherwise, the operation will be rejected. Note that one of A or B must always be a handle. Basic enforcement will support trusted subjects using input and output invulnerabilities as described earlier.

It is the responsibility of the constraint propagation phase, which is decoupled from basic policy enforcement, to maintain correct values of `current_min_lbl`. Note that constraint propagation is more complex, and involves a large number of objects and subjects simultaneously, and hence could be error-prone. Decoupling basic policy enforcement from the more complex constraint propagation means that we can have a higher level of assurance on its correctness.

To accommodate decoupling, overall policy enforcement operates as follows. The more complex policy checking, which includes constraint propagation, is invoked first. If this phase permits the operation, basic policy enforcement is invoked. The operation is denied if the first phase rejects the operation; otherwise, the operation is permitted.

2.2 Object Types and Operations

This section discusses the various objects for which our framework enforces integrity policies. Since the current implementation of our framework is for the Linux operating system, the descrip-

Object Type	Object Subtype	Operations to: Read/Modify object attribute: **bind**	**lookup**	**stat**	*unlink*	*rename*	**chmod etc.**	Create/Delete object: **create**	**delete**	Associate handles to objects: **open**	**close**	*listen*	**accept**	**connect**	Perform reads and writes: **read/recv**	**write/send**	**mmap**	*readfrom*	*sendto*
Files	File	+		+	+	+	+	+	+	+	+				+	+	+		
	Directory	+	+	+	+	+	+	+	+	+	+				+				
Links	Hard link	+	+	+	+	+		+	+										
	Symlink	+	+	+	+			+	+										
Volumes	File sys	+		+				+	+										
Pipes	Pipe			+				+			+				+	+			
	Named pipe	+	+	+	+	+	+	+	+	+	+				+	+			
Sockets	Unix Socket	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+		+	+
	Inet Socket	+		+	+			+		+	+	+	+	+	+	+		+	+
IPCs	Shmem			+				+	+	+	+								
	Other IPCs			+				+	+	+	+				+	+			

Table 2.1: Object types in Linux and the list of abstract operations available on them.

tion of the different objects and that of the operations on those objects is Linux specific in many ways. However the basic design is adaptive and can easily be mapped to any operating system.

2.2.1 Object types

While subjects and handles are largely homogeneous, there are many different types of objects that need to be considered. In order that operations on these objects be handled in a uniform way, we map the actual object operations into several abstract operations as shown in Table 2.1. For the purposes of policy enforcement, some of these operations are either ignored or are treated as a combination of other operations; such operations are shown in italics. The mapping of concrete operations to the abstract operations may not always be obvious for all object types, so we clarify this below:

- *Files:* We view creation operation as a combination of create and bind operations. The latter requires permission checks corresponding to the directory in which the object is being created. An unlink operation (an rmdir if it is a directory) is treated as a delete on the target object, while a rename is treated as a combination of delete and create.

  Directories are similar to files, and are handled the same as plain files in most cases, but there are some differences as well. For instance, they are not written or mmap'd, although they can be read. A lookup on a directory is treated the same as a read of the directory.

- *Hard links:* These are different from files because they do not have labels associated with them. Although our design could, in principle, associate labels with links, it would be difficult to implement: we rely on extended attributes for storing labels, but there is usually no support for associating extended attributes with links. As a result, permission decisions have to be made on the basis of labels associated with its parent (the directory in which the link resides) and its target (the file pointed by the link). In particular, link creation as well as removal are treated as a bind (to the parent directory) and a write to the target file.

- *Symbolic links:* Since symbolic links are stored as plain files (which contain the name of the target file), labels could be associated with them. Creation and deletion of a symbolic link are both treated as a bind on its parent, whereas a lookup is treated as a read of the link file (but

not the target). Symlinks need to be protected and it is possible that the symlink can have the security label different from that of the actual target. This stems from the fact that a symlink is a representation of its target in a *different context*. A possible attack scenario could be as follows:
Process A has write access to `/etc/passwd`. Now a malicious user creates a low integrity link to `/etc/passwd` in `/tmp`. Let's call this link as `/tmp/lowlink`. Now Process A may blindly write to each file in `/tmp/` because `/tmp` is a very commonly accessed file, and in this way it may destroy the contents of `/etc/password`. From PPI implementation point of view we have the following options:

1. We may choose not to enforce any policy for the symlink and basically just simply check the access permission of the subject on the actual target object. This sounds intuitively fine but has a pitfall. If this strategy is followed it would mean that Process A would essentially write onto `/etc/passwd`, when it never had the intent to do so. It simply wanted to write onto a file from `/tmp` directory. So this option does not work.

2. We treat the dereference of a symlink as a read operation and immediately downgrade Process A when it dereferences `/tmp/lowlink`.
This would mean that when the permission-check hook is invoked on `/etc/passwd` for the downgraded Process A, the access would be denied and the attack would be subverted. However this would cause a DOS attack situation for Process A because it could loose its ability to write to high integrity files in the future. So this strategy is also not good.

3. It's best to do a *virtual downgrade* of process A when it resolves a low level link (`/tmp/lowlink`). This way the attack is also subverted and Process A is not downgraded.

- *File systems:* The only operations on file system are mount and unmount. Note that a mount operation removes the existing interpretation of the mount point, and associates it with a new device. As such, mount is treated as a combination of a remove (of the original directory), a write to the device being mounted (unless it is a read-only mount), followed by a bind. Unmount is similar.

  In the case of mount/unmount operations, additional steps are needed for two reasons. First, the file system being mounted may not be trustworthy, and hence the labels provided by the file system may need to be overridden. Second, some file systems may not be capable of providing labels. To address these problems, we set the device label as the *maximum label* that is possible for any file within the file system represented by the device. In the first case, if the file system associates a label `l` with a file within it, then we take `glb_file_lbl(max_lbl,l)` as its label. The glb operation makes sense for integrity: it is the minimum of the integrity level of the file system and the specific label on a file. It also makes sense for confidentiality, since the glb will correspond to the maximum of the confidentialities of the entire file system and the specific file. (Here, `glb_file_lbl` is the natural extension of `glb_lbl` to `PPI_file_lbl`, where the greatest lower bound operation is applied to each component.) In the second case, we use `max_lbl` as the default label of all files in the volume. We may need a mount-time option by which the `max_lbl` is set to a value lower than that of `o`'s label.

- *Pipes:* As mentioned earlier, pipes and sockets differ from files in that they represent two

distinct object such that data written to one of them can be read from the other and vice-versa. As a result, create and open operations need to be interpreted differently, and appropriate handles associated with the objects.

Unnamed pipes can be created, but there is no way to delete them. They cannot be opened but can be closed. They cannot be bound to names, and hence do not support operations such as lookup, unlink, rename, chmod, etc. In contrast, a *named pipe* has a name in the directory tree, and hence supports all these operations. In particular, creation of a named pipe implies a bind operation, similar to plain files. Other name-related operations are also handled the same was as regular files (with the exception of how handles are associated with objects).

- *Sockets:* These are very similar to pipes. In particular, *Unix domain sockets* are very similar to named pipes, except for the following differences: (a) bind operation can be separated from creation, (b) handle to object associations are affected by additional operations (accept/connect), and (c) additional system calls to read/write are available (send/recv). (For datagram oriented sockets, sendto/recvfrom may also be used.)

  Internet-domain sockets differ primarily in terms of the addresses used for binding, and secondarily because LSM provides better hooks for mediating accept and connect system calls in the context of Internet-domain.

- *IPCs:* System V IPCs include those for manipulating message queues, semaphores, and shared memory.

  Shared memory needs to be treated differently because we cannot mediate interactions based on shared memory. Processes sharing memory can be handled as if they have a common mmapped file. Thus, a shared memory creation can be viewed as a combination of a file open (in read-only or read-write mode, based on how the shared memory segment is created), followed immediately by an `mmap`.

  Semaphores and *Message queues* are both handled in the same way, fairly similar to files.

2.2.2 Abstract operations

- `bind(PPI_object *ns, const char *nm, PPI_object o)`: We abstract all operations that associate an object `o` with a name `nm` within the namespace `ns`. A bind operation may be used while
  - *creating a new file, directory, or renaming one.* In this case `ns` will be a directory and `o` references either a plain file or a directory.
  - *creating a named pipe.* Identical to the previous case.
  - *binding a socket.* In this case, `ns` will refer to one of the predefined namespaces (e.g., `TCP_socket`) or a directory; and `nm` will be some string representation of its address.
  - *mounting a file system.* In this case, `ns` will be a directory, `nm` is empty, and `o` will refer to a device.

  In all cases, the bind will be permitted only if the label of the subject performing the bind is greater than or equal to that of the namespace where the new name is attached. In some cases, permissibility may depend on the value of `nm`, e.g., certain subjects may not be permitted to

bind to certain ports.

- `lookup(PPI_object *dir, const char* name)`: A lookup operation is treated as a read of a `dentry` structure. This means that the result of a lookup will have the label `glb(s->current_lbl, dir->current_lbl)`. If the result of a lookup is passed onto another lookup, as would happen with path lookups, this label is propagated. Note, however, that *we do not change the label of the subject as a result of lookups.* If and when the lookup results are used in an open operation, then the label of the file opened will be taken as the glb of the file's label and the dentry's label. As a result, a subject may be downgraded when it opens a file using a path name that has a low label, or if the file itself has a low label.

  Note that the decision not to downgrade subjects on lookups reflects an engineering trade-off. It could be argued that lookup operations, even when they do not feed into open operations, can affect subject behaviors in some cases to the point that its security will be compromised. (For example, consider a program that looks for file A, and if not found, reads file B.) But it appears that such a conservative view will lead to significant usability problems, while providing limited security benefits.

  In the current implementation, lookups leading to opens, too have not been implemented. The reason is that this may create usability issues for programs, that write to low integrity directories (like /tmp) very frequently and later read back from them. Lookups will cause downgrading of the subject when it tries to read back, what it had written earlier.

- `stat()`: A stat (or statfs, in the case of file systems) is treated similar to lookup, except that since the results of stat do not feed into an open, stats do not cause any information flow. The rationale is the same as with lookup.

- `unlink(PPI_object *ns, const char *nm, PPI_object o)`: It is treated as a combination of a `bind` (with the same arguments) and a delete operation on the object `o`.

- `rename(PPI_object *old_ns, PPI_object *new_ns, const char* new_nm, PPI_object *o)`:

  A rename operation is treated as a combination of an unlink operation on `old_ns`, and a create operation on `new_ns`.

- `create(PPI_object *ns, const char *nm)`: For all objects that have a name at the time of their creation, a create implies a bind as well, and hence bind-related enforcement as described above needs to be performed.

  A newly created object inherits its `current_lbl` from the `current_lbl` of the handle used in creation. (To determine the handle's `current_lbl`, rely on the invariant (13) on page 38 defining a write-handle's `current_lbl`, and the fact that a newly created write-handle inherits its `invul_lbl` from the subject's `discr_output_invul`.) The `min_lbl` field of the object is inherited from the subject's `discr_obj_min_lbl` field. The `read_log` and `write_log` fields of the newly created object are set to false.

  Note that an object gets a `PPI_object_lbl`, and not a `PPI_file_lbl`. If the object being created is a file object, then a `PPI_file_lbl` will need to be created for this file. The `obj_lbl` field of this object can obviously be initialized from the `PPI_object_lbl`. The

subj_lbl field will be initialized as follows:

  - is_super, read_log, write_log, can_handle_read_errors and can_handle_write_errors will be initialized as false.
  - current_lbl and invul_lbl are set to the current_lbl field of the object.
  - min_lbl and discr_obj_min_lbl are both set to min_lbl().
  - input_invul_lbl and output_invul_lbl can be set to min_lbl() or current_lbl value. (In reality, they don't matter as their values will be ignored: as per invariants (10) and (11) on page 38, invulnerability labels are honored only when the subject's current label is strictly greater than its invulnerability label, which is not the case here.)
  - discr_X field will be set to the same value as the X field for the remaining fields.

- delete(PPI_object *ns, PPI_object *o): A delete operation is treated as a write on o. This abstract operation is usually never invoked directly and is used in the implementation of unlink abstract operation to ensure that the subject calling unlink can establish a write handle on the object. In other words this operation simply checks if a subject can write to an object.
- open(PPI_object *o);
  - if the object being opened is instantiated from a file (or other entity that has an associated extended attribute that encapsulates a PPI_file lbl) then the object's labels are populated from the obj_lbl field of the PPI_file_label. Otherwise, the object must already exist (or has been just created), in which case its label is already populated.
  - A write open will first attempt to increase current_min_lbl of calling subject to the extent needed to satisfy the invariant (15) on page 38. If this fails, the open will be denied unless the write handle's can_handle_errors is true.
  - a read open will attempt to increase current_min_lbl of object being opened to the extent needed by invariant (14) on page 38. If this fails, the open will be denied unless the read handle's can_handle_errors is true.
  - a handle is associated with this object, and the labels associated with the handle are populated as given by the invariants in Section A.1.4. The new handle is added to the appropriate handle list maintained with the subject, and the object.
  - A read/write open is treated as two opens, one for reading, another for writing. Some objects, e.g., sockets and message queues, are implicitly read/write, some (e.g., pipe) are implicitly unidirectional, while other objects specify, at the time of open, whether a read/write/both is desired.
- close(PPI_object *o): No enforcement actions are required here, but the handles need to be cleaned up. In particular, a closed handle should be destroyed, and removed from the corresponding subject and object. However, LSM does not provide hooks on close operations except in the case where the associated object has only a single handle across all subjects. As a result, PPI has to deal with the possibility that some of its handles are stale. Before denying any operation, PPI needs to ensure that handles are not stale. Otherwise, operations that could be permitted without violating information flow policies may be rejected.

- `listen`: No PPI checks are required on listen.
- `connect` on **connection-oriented** sockets: In this case, the object involved is a socket O_{sock}, and it must have two associated handles H_r and H_w belonging to the subject (which is a client). At the point of connection, the following need to be done:
  - Create new object O_{accept} inside the server subject S.
  - Create two new handles H'_r and H'_w on S. H'_r will be initialized with the value of H'_{rl}, the read handle associated with the socket O_{listen} on which the `accept` call was made.
  - Make the following associations between handles and objects:
    * H_r with O_{sock}
    * H_w with O_{accept}
    * H'_r with O_{accept}
    * H'_w with O_{sock}

    While making these associations, all the invariants on all of the handles, objects and subjects mentioned above should be satisfied, or be satisfiable by making allowable changes to `current_lbl` and `current_min_lbl` of the entities involved. (The list of entities involved will be a superset of the handles, objects, and subjects mentions above.) If not, the connect request should be denied. Naturally, none of the changes mentioned above should be committed before this check is made.

  Note that, although some of these steps are not evident in Figure 2.1, they are fully consistent with that picture.
- `accept`: No checks can be performed when accept is called, since the client identity is unknown at this point. If the client is on the same OS, then, as described under `connect` operation above, we can enforce accept-related policy in the connect hook. When the client is remote, in order to enforce any policy, we need an appropriate LSM hook that allows examination of client credentials, and to fail the accept call if the credentials are insufficient. Unfortunately, such hooks are not yet available, so PPI cannot enforce any policies on which clients can connect to it. In particular, this means that if a remote client has low integrity, then the server will be downgraded, and mechanisms such as `min_lbl` setting on the server cannot prevent such downgrading.[1]
- `connect` on **datagram** sockets: Let H_r and H_w be the handles associated with the socket object O_{sock} on the client subject C. Let O_{server} be the server-side socket that corresponds to the server address specified in `connect`. Perform the following associations:
  - H_r with O_{server}
  - H_w with O_{sock}

[1]This raises the question: why bother enforcing any policies on accept if we can only do this locally, and cannot prevent remote connections from unauthorized clients? The reason is that for remote connections, we have alternative mechanisms for keeping unauthorized clients out, such as a firewall. Although firewall mechanisms such as iptables can enforce policies on connections by local clients, these policies are based on uids rather than labels. As a result, unless we enforce policies on accept, malicious local applications can mount trivial DoS attacks by connecting to servers and degrading them.

As before, all invariants need to be checked, and if they cannot be satisfied, connection should be denied.

Since datagram connect has no real server-side effects, no associations are made on the server side. Any effect of communication will be reflected when the server reads the data sent by the client.

- `read(PPI_handle *h)`: If `current_lbl` of the read handle, as given by invariant (12) on page 38, is greater than or equal to the `discr_min_lbl` of the subject, then the read will be permitted. The subjects `current_lbl` is set to that of the handle. If this represents a change to the subject's `current_lbl`, then the new label value needs to be propagated using the rules described earlier.
- `readfrom`: There is no way to find the source of the message until the data has already been read. So, no checks can be done here. Instead, we will rely purely on a sender-side check. (The situation is similar to that of accept on connection-oriented sockets, and the resolution is also similar.)
- `write(PPI_handle *h)`: If `current_lbl` of the write handle, as given by invariant (13) on page 38, is greater than or equal to the `min_lbl` of the object, then the write will be permitted. The objects `current_lbl` is set to that of the handle. If this represents a change to the object's `current_lbl`, then the new label value needs to be propagated using the rules described earlier.
- `sendto`: This operation should be treated as a combination of a datagram connect and a write, and the corresponding checks/propagations performed.
- `mmap(PPI_handle *h)`: Set `is_flow_mediated` field of the handle to `false`.

Chapter 3

Framework Implementation

This chapter gives a glimpse of some of the aspects of the implementation of the PPI Framework, the challenges faced and the solutions provided.

3.1 PPI Framework Development

The PPI Framework has been implemented for the Linux operating system (*Ubuntu 8.04 LTS*). The default kernel version (*version 2.6.24*) has been patched using the standard patch [15] to reverse the conversion of the Linux Security Module (LSM) Framework to static interface. Hence the PPI Framework has been developed as a loadable kernel LSM module.

The LSM framework provides hooks to mediate system calls and system operations pertaining to inodes, files, tasks, semaphores, shared memory, sockets, and message queues. The abstract operations discussed in Section 2.2.2 have been mapped to these hooks after carefully examining the sequence of invocation of these hooks and also taking into consideration whether the hook provides the necessary information/parameters which is/are necessary for the correct functioning of the corresponding PPI abstract operation. Details of the mapping of these operations to the LSM hooks have been discussed in Section A.2, along with the corresponding kernel flow diagrams.

3.2 Framework Hooks

This section broadly classifies the hooks of the PPI framework based on their purpose. A short description of each hook has also been provided. LSM provides many more hooks than the ones discussed here. However only the pertinent hooks have been used in our implementation, the criteria for selection being the hook's appearance in the sequence of invocation of related hooks, the hook's parameters and the hook's return type.

Much of the work performed in each of the selected hooks falls into the following categories:

- Updating the data structures in response to various operations on subjects and objects; and maintaining the invariants listed in the appendix after each such operation.
- Storing information that is available in one LSM hook so that it can be used in a subsequent hook where it is needed; and more generally, reconstructing information needed by our framework that is not directly available in the LSM hooks.

- Enforcing integrity policies on objects and subjects and making access decisions. It is important to note that a hook with a *void* return type cannot be used for making access decisions.

Section A.2 discusses the hooks and the mapped abstract operations in greater detail. The classification of hooks used in our framework is the following:

- **Security hooks for program execution operations**
  - `ppi_bprm_check_security` : This hook is for checking if the subject can read the binary and execute it without violating the integrity invariants.
  - `ppi_bprm_apply_creds` : This hook updates the integrity label of the running task by considering the binary's `file_label` and its `object_label`.
- **Security hooks for filesystem operations**
  - `ppi_security_sb_mount` : This hook checks if the runtime binding of a device can occur with a mount point.
  - `ppi_security_sb_unmount` : This hook simply checks if the subject can unmount a device.
- **Security hooks for inode operations**
  - `ppi_inode_alloc_security` : This hook is used to allocate an in-memory object security structure to every object represented by an inode and assign a label to it.
  - `ppi_inode_free_security` : This object de-allocates the object security structure and cleans up the memory allocated for its label and handles (if the handles were not already closed).
  - `ppi_inode_init_security` : This hook makes the object security structure, associated with the inode, persistent, by writing it on the persistent media (disk), typically in the inode's extended attribute space.
  - `ppi_inode_create` : This hook is specifically for regular files and helps the framework perform regular-file specific permission checks (such as `ppi_bind`).
  - `ppi_inode_link` : This hook is specifically for hard-links and helps the framework perform hard-link specific permission checks (such as `ppi_bind`).
  - `ppi_inode_unlink` : This hook helps the framework perform permission check (such as `ppi_unlink`) on the inode, to remove hard links to it.
  - `ppi_inode_symlink` : This hook is specifically for symbolic-links and helps the framework perform symbolic-link creation checks (such as `ppi_bind`).
  - `ppi_inode_mkdir` : This hook is for directories and helps the framework perform directory creation checks (such as `ppi_bind`).
  - `ppi_inode_rmdir` : The framework uses this to check if a directory can be unlinked from its parent namespace.
  - `ppi_inode_mknod` : This hook deals with permission checks for creation of special files like pipes and named sockets.
  - `ppi_inode_rename` : This hook primarily implement the abstract operation `ppi_rename`.

  - `ppi_inode_follow_link` : This hook is used for maintaining link traversal information which is used for implementing *virtual downgrades*.
  - `ppi_inode_permission` : This hook performs the handle creation operation by checking the mode in which the inode is being accessed. The abstract operation `ppi_open` is performed in this hook.
  - `ppi_inode_setattr` : This hook checks permission before setting file attributes.
  - `ppi_inode_getattr` : This hook checks permission before getting file attributes.
  - `ppi_inode_delete` : This hook can be used to release any persistent label associated with the inode. Currently this hook is not being used because the clean-up is performed in `ppi_inode_free_security`.
  - `ppi_inode_setxattr` : This hook checks permission before setting the extended attributes.
  - `ppi_inode_getxattr` : This hook checks permission before getting the extended attributes.
  - `ppi_inode_removexattr` : This hook checks permission before removing the extended attributes from persistent media.

- **Security hooks for dentry operations**
  - `ppi_d_instantiate` : This hook is invoked whenever a dentry structure is instantiated for an inode, in the d_cache.

- **Security hooks for file operations**
  - `ppi_file_permission` : This hook is invoked for every read and write attempted on a file object. Our framework calls the `ppi_read` and `ppi_write` abstract operations depending on the mode the file is being accessed.
  - `ppi_file_free_security` : Our framework uses this hook for cleaning up the handles associated with the file object.
  - `ppi_file_ioctl` : This hook checks permission for an ioctl operation on file.
  - `ppi_file_mmap` : This hook checks permissions for a mmap operation. Reads and rites to the mmap'ed region are unmediated, this hook helps the framework in setting the desired flag for the read and write handles to the mmap'ed region.
  - `ppi_file_fcntl` : This hook checks permission before allowing the file operation specified by the `cmd` parameter from being performed on the file.

- **Security hooks for task operations**
  - `ppi_task_create` : This hook is used by the framework to differentiate between fork and clone events.
  - `ppi_task_alloc_security` : This hook is used to assign subject security structure to the task in a system.
  - `ppi_task_free_security` : This subject performs the clean-up of the subject security structure.
  - `ppi_task_setrlimit` : To perform permission checks on the subject which tries to modify resource limits.

- `ppi_task_kill`: To perform task permission check on one task which is trying to send a signal to another task.

- **Security hooks for Unix domain networking**
  - `ppi_socket_unix_stream_connect` : Checks permissions before establishing a Unix domain stream connection.
  - `ppi_socket_unix_may_send` : Checks permissions before connecting or sending datagrams from one socket to another.

- **Security hooks for socket operations**
  - `ppi_socket_create`: Checks permissions prior to creating a new socket.
  - `ppi_socket_bind` : Checks permission before socket protocol layer bind operation is performed and the socket is bound to the specified address.
  - `ppi_socket_connect`: Checks permission before socket protocol layer connect operation attempts to connect socket to a remote address
  - `ppi_socket_listen`: Checks permission before socket protocol layer listen operation.
  - `ppi_socket_accept`: Checks permission before accepting a new connection. However from the framework perspective, we don't really do anything in this hook. Details on this are covered in Section 3.3
  - `ppi_socket_post_accept` : This hook allows our framework to copy security information into the newly created socket's inode. However this hook too is left un-implemented.
  - `ppi_socket_sendmsg` : Checks permission before transmitting a message to another socket.
  - `ppi_socket_recvmsg` : Checks permission before receiving a message from another socket.

- **Security hooks for System V IPC Message Queues**
  - `ppi_msg_queue_associate`: Checks permission when a message queue is requested through the `msgget` system call.
  - `ppi_msg_queue_msgctl`: Checks permission when a message control operation specified by `cmd` is to be performed on the given message queue
  - `ppi_msg_queue_msgsnd` : Checks permission before a message is enqueued on the message queue.
  - `ppi_msg_queue_msgrcv` : Checks permission before a message is dequeued on the message queue.

- **Security hooks for System V Shared Memory Segments**
  - `ppi_shm_associate`: Checks permission when a shared memory region is requested through the `shmget` system call.
  - `ppi_shm_shmctl`: Checks permission when a shared memory control operation specified by `cmd` is to be performed on the shared memory region.
  - `ppi_shm_shmat` : Checks permissions prior to allowing the `shmat` system call to attach the shared memory segment to the data segment of the calling process.

- **Security hooks for System V Semaphores**
  All operations performed in the following hooks are exactly the same as those performed for the corresponding hooks for System V Shared Memory Segments.
  - `ppi_sem_associate`
  - `ppi_sem_semctl`
  - `ppi_sem_semop`

3.3 Challenges and Solutions

The greatest challenge, we faced, while developing this framework, was mapping our abstract operations to the LSM framework. This required deep understanding of the flow of kernel code related to all the hooks of the LSM framework and especially that of the hooks mentioned in Section 3.2. There were some shortcomings in the LSM framework which proved to be particularly difficult to deal with. Some of these are enlisted below.

- LSM does not provide a hook to track the decrement of a file structure's reference count. A `file` structure in the file descriptor table of a process indicates that the file is opened by that process. If the file is shared (case of `fork`), its reference count is simply incremented. `ppi_file_free_security` is invoked only when all references to `file` structure drop to zero. Our framework maintains handles per file descriptor. When a process forks a child process, our framework replicates the PPI handles for the forked process because the forked process too has a file descriptor corresponding to each shared file in its file descriptor table. Now, if either the parent process or its child (but not both) call `close` on a shared file, our framework is unable to close PPI handles for that process. The `file` structure for the shared file remains non zero and therefore `ppi_file_free_security` is not invoked till all processes call `close` on the shared file. This leads to accumulation of stale handles in the system. Stale handles typically affect usability: the `current_min_lbl` of a subject could continue to remain high, even when it closes a write handle (whose creation may have been responsible for the high value of the subject's `current_min_lbl`), which may eventually result in a permission-denial while creating a new read handle on an object with lower `current_min_lbl`.

  For solving this problem we employ a strategy to validate each handle before it is used, so that stale handles could be closed as soon as they are detected. For this we maintain information such as an inode's inode-number, generation-count and an object's access mode in the `PPI_handle` data structure. This information is verified during the validation phase.

- Our framework maintains read and write handles for each socket that is created in the system. These handles are created at the time of socket creation.
  LSM provides the `ppi_socket_create` hook to mediate the socket creation event. At the time of a connection `accept` (for connection-oriented stream sockets), the Linux kernel creates a new socket to handle the accepted connection, which is different from the listening socket. However the creation of this new socket goes unmediated because the kernel does not invoke `ppi_socket_create` hook at the server end. Consequently, the framework is unable to create PPI handles on the new socket. Is is not possible to create these handles in the `ppi_socket_accept` hook because the new socket structure (`struct sock`) is not com-

pletely populated for use. Also the client identity at the time of `ppi_socket_accept` is unknown. Even though the new socket structure is populated in `ppi_socket_post_accept`, PPI handle creation and permission checking cannot be performed in this hook because the return type of this hook is `void`, and it therefore cannot return success or failure. Thus handle creation and permission checks (constraint propagation checks) need to be deferred to `ppi_socket_sendmsg` and `ppi_socket_recvmsg`. The drawback of this approach is that a constraint propagation failure in any of these hooks does not lead to connection termination. The situation results in delayed failures and might affect usability of applications.

3.4 User Interfacing via Securityfs

User interfacing with the PPI kernel module is necessary for providing the initial labels to the subjects/objects, modifying labels at runtime and for setting handle-specific discretionary values by PPI aware applications.The PPI interface management is done using **Securityfs**. Securityfs is a special-purpose virtual filesystem, meant to be used by security modules, some of which otherwise create their own filesystems. It should be mounted on `/sys/kernel/security`. Securityfs thus looks, from user space, like part of sysfs, but it is a distinct entity. The support for securityfs is enabled by default in all new Ubuntu distributions, starting from Ubuntu 7.10. For interfacing PPI using securityfs, no extra kernel patch is required. Securityfs was chosen for interfacing because of the following considerations:

- *procfs* was never meant for interface control but rather for process related statistics and control.
- Securityfs provides a flexible framework for defining module-specific methods for handling securityfs files. Typical examples are read and write methods for input files and configuration files respectively.
- It provides homogenity with other LSMs like Apparmor and Tomoyo. SELinux however has its has own filesystem called *selinuxfs*.
- Using securityfs instead of ioctl calls, saves the overloading of ioctl() method.

PPI utilizes the Securityfs feature of being able to define file specific methods, which make the file behave as desired. The user of the PPI interface can either pass the name of the object (file/socket/pipe) or its descriptor (resulting from an *open* system call) for setting or reading PPI labels. The Securityfs files that the PPI module registers can not be deleted from user space, therefore no explicit security needs to be ensured for these files.

Chapter 4

Evaluation

This section describes the evaluation of the PPI framework. Evaluation work for our implementation is divided into two parts. The first part deals with the correctness of the framework, where we develop scenarios to test the working of our algorithms and validate the correctness of our technique. The second part deals with measuring the performance overhead in a system running with our LSM module. All evaluations have been done on a VMware virtual machine with 2.6 GHz single core processor, 512 megabytes of RAM and 10 GB of free hard disk space.

4.1 Evaluation of Correctness

To test the correctness of our system we developed more than 50 use-cases and wrote programs to validate the behavior of our framework for each scenario. The label propagation and constraint propagation occurred just as we expected, with our framework denying and permitting accesses in line with our expectations. Some of the use-cases have been enlisted below.

- An attempt to read a low integrity file by a subject gets denied, if the subject is writing to a high integrity file.
  In our experiment, the subject had `current_lbl` = 7 and `current_min_lbl` = 0. It then opens a file (`current_lbl` = 7 and `current_min_lbl` = 6) in write-only mode. This results in increasing the subject's `current_min_lbl` to 6. The subject then attempts to open a file (`current_lbl` = 5 and `current_min_lbl` = 4) in read-only mode which results in increasing the file's `current_min_lbl` from 4 to 6. The read access is denied because the integrity invariant gets violated for the file on which the read was attempted.
- An attempt to create a directory/file/named-pipe in a directory D_1 is denied by `ppi_bind` if the `current_lbl` of the subject is less than the `current_lbl` of D_1.
- A high-integrity subject attempting to open a file via a low-integrity symbolic link gets *virtually downgraded* to the `current_lbl` of the symbolic link. In our experiment the subject was made to traverse 3 symbolic links in the lookup operation to open the target file. As expected, the subject was downgraded to the least of the `current_lbl` values of the 3 symbolic links.
- We developed scenarios where subjects forked child processes and each of those child processes invoked `close` on some of the files that they shared (as a result of the `fork` operation)

with their parent process. This resulted in multiple stale handles in the system. We then initiated new open operations by the forked processes. The constraint propagation phase, then validated all handles before using them. We observed that all stale handles were either closed successfully or re-used for another object.

4.2 Evaluation of Performance

This section discusses the performance evaluation that we did for our framework. We used the standard test-suite for the core-utils 6.10 in conducting our experiments. Our framework passed all tests of the test-suite. A significant overhead of 30% was introduced by our framework. The overhead was in the CPU time taken to execute the tests. Table 4.1 compares the CPU time taken for running the core-utils test-suite, with and without the PPI-framework module.

	Time in Seconds	
	Without PPI Framework	With PPI Framework
chgrp	1.144	1.58
chmod	1.896	2.348
chown	0.688	0.868
cp	7.564	10.281
cut	1.496	1.872
ls	2.692	3.492
mkdir	2.112	3.02
rmdir	0.492	0.652
wc	0.404	0.592
dd	0.788	1.064
head	0.732	1.048
install	0.672	0.912
join	0.5	0.692
ln	0.852	1.292
pr	3.06	4.044
readlink	1.068	1.532
sort	1.112	1.4
tail	1.024	1.316
touch	0.952	1.184
tr	1.268	1.688
uniq	1.612	2.088
rm	6	9.257
mv	5.068	6.456

Table 4.1: Timing Results with Coreutils-6.10

Each timing result for each core utility in Table 4.1 is an average value of 3 test runs. The average overhead in CPU time-taken is around 30%. Our implementation can be further optimized and we expect to reduce the overhead after the optimizations have been done.

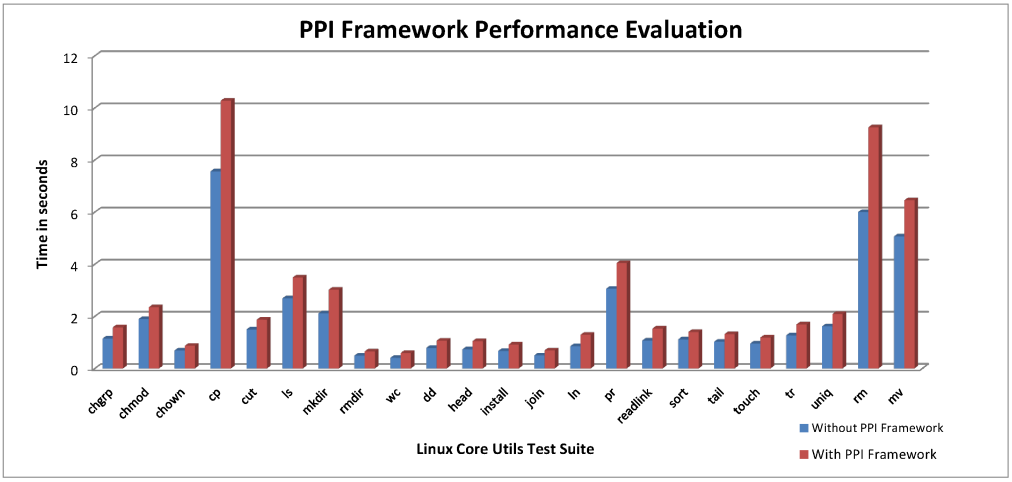

Figure 4.1: PPI-Framework Performance Comparison for Table 4.1

Figure 4.1 gives a graphical comparison of our test results. It can be observed that the time taken, with our framework module, for core utilities such as `cp`, `mv`, `rm`, `mkdir` and `rmdir`, is particularly higher than the time taken for these utilities without our framework module. The reason for this is because our framework performs extra checks for namespace binding (`ppi_bind` abstract operation) for each of these utilities. In addition to these bind related checks, the constraint propagation checks are performed as usual. As a part of the future work, discussed in Chapter 6, we intend to reduce the overhead in constraint propagation, which will automatically reduce the overall CPU time.

Chapter 5

Related Work

This chapter discusses in brief, some of the works that are related to our research. This chapter also mentions about those works that have served as guidelines for the development of our framework.

An early, and significant work on information-flow based techniques is the Biba integrity model [4] which serves as a guiding principle for similar works. However Biba model is very strict and simply denies all *write-ups* and *read-downs*, thus it tends to break the functionality of many applications and suffers from poor usability.

Low Watermark model [5, 11] builds over the Biba model by adding a *subject downgrade* policy. This means that a subject gets downgraded to a lower integrity level when it reads from a low integrity object, and subsequently runs at a reduced integrity level for the remainder of its lifespan.

Another work, IX [9] aims at developing an experimental multi-level security model for the Linux operating system. This technique uses dynamic labels for subjects and objects, and tracks information flow for providing confidentiality and integrity. However it does not provide a choice of policy enforcement, like our framework does, and also does not decouple policies from labels, which is one of the key features of the PPI framework.

Back to the Future system [7] enforces only the *no read down* (reads that occur from low-integrity sources) policy. The advantage of such a model is that it can thwart an attempt by malware to inject itself into inputs consumed by benign applications, as demonstrated by their experimental results. However, this scheme too is not complete, because, an attempt to use the output of an untrusted application, requires user intervention. User responses to such prompts have statistically been proven to be unreliable, from a security standpoint. Moreover these prompts leave too much to the user's judgment and can overwhelm the user. Also this approach suffers from the problem of delayed detection wherein malware actions aren't thwarted at the point where they overwrite critical files, but at the point where a benign application uses them. In contrast to this approach, our framework does not involve user interaction for making policy decisions and detects writes performed by the malware, much earlier.

Unlike the *Back to the Future* model, Windows Vista Security Model does not have security enforcement policies for *read-downs* and only enforces policies that mediate and thus prevent

write-ups (writes that occur on high-integrity objects, whose integrity must be preserved). This design decision has been made for avoiding usability issues. This model is quite vulnerable to attacks, as is suggested by a real world attack: the attacker simply overwrites the Vista start-menu file, an action that does not involve escalated privilege level. The affected start-menu items now points to entry-points in the malware rather than the usual applications. However, the naive user is oblivious of this and trusts the start-menu entries. Thus when he clicks on a menu item and is prompted by the Vista UAC, which requests higher privilege level for running the intended program, the user almost certainly allows the escalation of privileges. The malware then executes with escalated privileges thereby compromising system security.

Safe Execution Environments [2, 12, 13, 17] employ isolation techniques to confine untrusted applications. The same technique is used by virtual machines [3, 16]. The main drawback with isolation techniques is that maintaining multiple isolated work environments is not feasible from a user's perspective. An input file required by one of the untrusted applications in a specific isolated work environment needs to be explicitly copied into that environment by the user.

Several *sandboxing* techniques [1, 6, 10] have been developed. However development of sandboxing policies can turn out to be quite challenging because of the ease of multi-step attacks, as described in the Chapter 1.

SELinux [8] uses domain and type enforcement for confining applications. The primary focus is on servers and it aims at developing policies that enforce least privilege principles on the related application. However the applications, for which SELinux generates policies, are trusted and therefore the policies developed for these trusted applications cannot be enforced for untrusted applications.

Chapter 6

Conclusions and Future Work

In this thesis we presented a framework for enforcing information flow policies. Our approach of propagating integrity constraints for maintaining integrity invariants promotes early failures and hence improves usability of applications. The framework decouples the policy enforcement from the labels assigned to objects and subjects in the system. Our framework enforces integrity policies on all types of objects (files, sockets, pipes, IPC channels, directories, devices), thus making our solution complete in terms of object coverage. We provide mechanisms for limiting the trust of a process by controlling its invulnerability level. Our framework provides flexibility to integrity-aware applications to set discretionary attributes on the application-specific handles, maintained by our framework.

Our framework implementation, for the Linux operating system uses the LSM hooks and shows that our framework is practical. The overheads incurred by our framework can be reduced, in future, by further optimization of our techniques. Our documentation of the LSM infrastructure and our flow diagrams serve as a good reference for someone developing a LSM module such as ours.

We intend to extend our implementation for enforcing policies on internet sockets and IPC channels such as message queues, shared memory and semaphores. Once the completeness of object coverage is ensured by the implementation, we intend to conduct further experiments to evaluate the correctness and performance of our framework. Further evaluation results, would help us analyse the potential modules of our framework that could be optimized for an overall reduction in the overhead incurred by the current implementation.

We also intend to extend our framework for enforcing information flow policies, not just for integrity but also for confidentiality. Since integrity and confidentiality are largely orthogonal, we could easily modify our label design to model a label as a linear lattice, with the top element corresponding to highest integrity and lowest confidentiality; and the bottom element corresponding to lowest integrity and highest confidentiality.

Appendix A

Data Structures and LSM Mappings

This appendix shows the low level details of the key data structures used by PPI-Framework.

A.1 Key Data Structures

A.1.1 Labels

The most basic data structure is a *Label*: it is an abstract data type that provides the operations listed below.

```
struct PPI_lbl {
  unsigned char level;
};
```

Operations.

```
PPI_lbl max_lbl();
PPI_lbl min_lbl();

PPI_lbl lub_lbl(PPI_lbl, PPI_lbl);
PPI_lbl glb_lbl(PPI_lbl, PPI_lbl);

bool geq_lbl(PPI_lbl, PPI_lbl);
bool gt_lbl(PPI_lbl, PPI_lbl);
bool eq_lbl(PPI_lbl, PPI_lbl);

const char *serialize_lbl(PPI_lbl);   /* Serialize a label */
PPI_lbl new_lbl(const char*); /* De-serialize a label */
```

Object Labels

```
struct PPI_object_lbl {
  bool read_log, write_log; // Enable/disable object logging.
  PPI_lbl current_lbl;      // Object's current integrity label.
  PPI_lbl min_lbl;          // current_lbl >= min_lbl.
  PPI_lbl current_min_lbl;  // current_lbl >= current_min_lbl.
```

```
  PPI_lbl new_min_lbl;      // Used internally in the implementation.
};
```

Operations.

```
const char *serialize_lbl(PPI_object_lbl); /* Serialize a label */
PPI_object_lbl new_object_lbl(const char*); /* De-serialize a label */
```

Invariants.

- `current_lbl >= new_min_lbl >= current_min_lbl >= min_lbl`

Notes.

- When a new object is created, it inherits the `current_lbl` of its associated handle. Its `min_lbl` and `current_min_lbl` are set from the `discr_obj_min_lbl` of the subject. Its `read_log` and `write_log` fields are also set from the corresponding fields of the subject. Object's `current_min_lbl` takes into account the policies imposed by readers of this object.
- The value of `current_min_lbl` is updated incrementally by the implementation whenever there is a change to any of the quantities in the right-hand side of the invariant above. This recomputation relies on a two-phase protocol. During a run of this protocol, `new_min_lbl` is used to remember an intermediate value of `current_min_lbl`.

Subject Labels

```
struct PPI_subject_lbl {
  bool is_super;             // Enable arbitrary label changes.
  bool read_log, write_log;  // reads/write subject logging.
  bool can_handle_read_errors, can_handle_write_errors;

  PPI_lbl current_lbl;       // Subject's current label.
  PPI_lbl min_lbl;           // current_lbl >= min_lbl.
  PPI_lbl discr_min_lbl;     // current_lbl >= discr_min_lbl.
  PPI_lbl discr_obj_min_lbl;

  PPI_lbl invul_lbl, discr_invul_lbl;
  PPI_lbl input_invul, output_invul;
  PPI_lbl discr_input_invul, discr_output_invul;
  PPI_lbl virtual_current_lbl;   // Used while symlink traversal
  PPI_lbl current_min_lbl;   // current_lbl >= current_min_lbl.
  PPI_lbl new_min_lbl;       // Used internally in the implementation.
};
```

Operations.

```
const char *serialize_lbl(PPI_subject_lbl);
PPI_subject_lbl new_object_lbl(const char*); // deserialize
```

The purpose of most fields in the above data structure we explained earlier. One exception is `discr_obj_min_lbl`, which has been introduced to provides a means for a subject to decide the `min_lbl` for newly created objects. By default, its value will be `min_lbl()`. Also

`current_min_lbl` takes into account the policies imposed by readers of this subject.

Invariants.

- `current_lbl >= new_min_lbl >= current_min_lbl >= discr_min_lbl >= min_lbl`
- `discr_X_lbl >= X_lbl`, where `X` is one of `invul` or `input_invul`
- `discr_output_invul_lbl <= output_invul_lbl`
- `is_super` $\Rightarrow$ `current_lbl == max_lbl()`

Notes.

- The only fields that a subject can change are the discretionary fields (i.e., fields containing the word `discr_`.)
- In addition, a subject S can request its current label to to be downgraded using a PPI API call `downgrade_me(IntegrityLabel l)`.
- A super subject (i.e., a subject with the `is_super` attribute set) can change any attribute of other subjects; but this is not meant to be done, as the changes may violate invariants and hence cause unpredictable problems. Hence such changes are to be attempted only as an absolute last resort.

File Labels

Files stored on the disk may be used as an object (when they are read or written) or as subjects (when they are executed). Thus, they have both labels.

```
struct PPI_file_lbl {
  PPI_object_lbl obj_lbl;
  PPI_subject_lbl subj_lbl;
};
```

Operations.

```
const char *serialize_lbl(PPI_file_lbl); /* Serialize */
PPI_file_lbl new_file_lbl(const char*); / * De-serialize */
```

Invariants. The following are additional invariants that apply over and above the invariants on `PPI_object_lbl` and `PPI_subject_lbl`.

- `X.current_min_lbl == X.new_min_lbl == X.min_lbl` where `X` stands for either `obj_lbl` or `subj_lbl`.
- `subj_lbl.discr_X_lbl == subj_lbl.X_lbl`, where `X` is one of `invul`, `min_lbl`, `input_invul` or `output_invul`

Notes.

- Given a subject S and a file F such that S`.current_lbl` $\geq$ F`.obj_lbl.current_lbl`, S can modify all the label fields of F subject to the above invariants, with the exception of: F`.subj_lbl.is_super`, F`.subj_lbl.invul_lbl`, F`.subj_lbl.input_invul`, and F`.subj_lbl.output_invul`. These four fields can be changed only if S`.is_super`

is true.

- A super subject can change any component of a `PPI_file_lbl`. The implementation must check the invariants after any such change, and ensure that all of them will hold.
- Note that `discr_obj_min_lbl` field of subject labels will be stored on disk, as it provides the only way to control the default `min_lbl` of objects created by a subject.
- In our implementation, file labels will be stored using extended attributes supported by the underlying file system. In particular, serialize/deserialize operations will be used to covert file labels into strings (or vice-versa), and these strings will be stored as extended attributes.
- Note that the serialize operation on files can choose to omit fields whose values can be obtained from other fields and the invariants above.

Handle Labels

```
struct PPI_handle_lbl {
  PPI_lbl invul_lbl;
  PPI_lbl discr_min_lbl; // Applicable only for read handles
};
```

A.1.2 Object

```
struct PPI_object {
  const PPI_object *volume; // Object's File-system Label
  PPI_object_lbl label;
  List *read_handles;
  List *write_handles;
  struct inode *inode; //pointer to corresponding inode
};
```

A.1.3 Subject and SubjectGroup

```
struct PPI_subject_group {
  int reference_count;    // Same as the number of subjects in group.
  PPI_subject_lbl label;
  PPI_handle **fd2rhandle; // Maps fd numbers to PPI_handle pointers,
  PPI_handle **fd2whandle; // They are allocated at subject creation.
  List *read_handles;      // All handles stored in these lists,
  List *write_handles;     // incl. those stored in fd2handle arrays
};

struct PPI_subject {
  bool is_same_group; // Internal use: store info across LSM hooks
  bool is_named_pipe; // Internal use: store info across LSM hooks
  PPI_subject_group *group;
  struct task_struct *task; // Pointer to corresponding Linux task.
};
```

Notes. It is important to note that the fd information associated with a handle is not reliable. First, due to structure of hooks, fd information is unavailable at the time of handle creation. Sec-

ond, when dup or dup2 are used, we do not have any hooks to track them. Third, fds may get closed, but we don't always get notified. As a result, the following can happen:

- We do not have an fd for a handle.
- There is an fd for which we do not have a corresponding handle, e.g., due to `dup`.
- We have an fd for a handle, but this is incorrect as a result of the subject executing a `dup2` (or a combination `dup` and `close`).
- We have a handle but the corresponding fd has already been closed.

A.1.4 Handles

```
struct PPI_handle {
  bool can_handle_errors;
  bool is_flow_mediated;
  bool handle_type;       // read = 0, write = 1.

  PPI_handle_lbl label;
  PPI_object  *obj;
  PPI_subject *subj;

  int inode_num;          // These fields are used to associate and
  struct file *file_ptr;  // validate handles with file structures
  int fd;                 // maintained by the kernel.

};
```

Operations.

```
bool read_handle(const PPI_handle*);
bool write_handle(const PPI_handle*);
PPI_lbl invul_lbl(const PPI_handle*);
current_lbl(const PPI_handle*);
current_min_lbl(const PPI_handle*);
```

Invariants. Below, `H` stands for any handle, `RH` for any read handle and `WH` for any write handle.

1. `member(RH, RH.obj->read_handles)`
2. `member(WH, WH.obj->write_handles)`
3. `member(RH, RH.subj->group->read_handles)`
4. `member(WH, WH.subj->group->write_handles)`
5. `invul_lbl(H) == H.label.invul_lbl`
6. `invul_lbl(RH) >= RH.subj->input_invul_lbl`
7. `invul_lbl(WH) <= WH.subj->output_invul_lbl`
8. `RH.can_handle_errors` $\Rightarrow$
   `RH.subj->can_handle_read_errors && RH.is_flow_mediated`
9. `WH.can_handle_errors` $\Rightarrow$
   `WH.subj->can_handle_write_errors && WH.is_flow_mediated`

10. `invul_lbl(RH) < RH.subj->current_lbl` $\Rightarrow$
    `RH.subj->current_lbl > RH.subj->invul_lbl`
11. `invul_lbl(WH) > WH.subj->current_lbl` $\Rightarrow$
    `WH.subj->current_lbl > WH.subj->invul_lbl`
12. `current_lbl(RH) == (RH.obj->current_lbl < invul_lbl(RH) ?`
    `RH.obj->current_lbl : RH.subj->current_lbl)`
13. `current_lbl(WH) == max(WH.subj->current_lbl, invul_lbl(WH))`
14. `current_min_lbl(RH) == min(discr_invul_lbl(RH),`
    `max(RH.subj->current_min_lbl, discr_min_lbl(RH)))`
15. `current_min_lbl(WH) == (WH.obj->current_min_lbl > invul_lbl(WH) ?`
    `WH.obj->current_min_lbl : WH.subj->current_min_lbl)`
16. `!RH.can_handle_errors` $\Rightarrow$ `RH.obj->current_min_lbl >= current_min_lbl(RH)`
17. `!WH.can_handle_errors` $\Rightarrow$ `WH.subj->current_min_lbl`
    `>= current_min_lbl(WH)`

Notes.

- Invariants 1 through 4 capture the requirement that cross-linking information pertaining to handles, objects and subjects be consistent.
- We do not have an explicit invariant `discr_min_lbl(RH) >=RH.subj->min_lbl`. This is because the invariant (14) on page 38 will already ensure that the label of a read handle will remain high enough to support the subject's `min_lbl`.
- Invariants 5 through 11 capture constraints between various invulnerability-related labels of handles and their associated subjects. In essence, they state that the extent of invulnerability cannot be greater than that of the subject.
- Note that `current_lbl` flows from a read handle to a subject on the next read operation, if `is_flow_mediated` is set; otherwise, it flows immediately.
- Similarly, `current_lbl` flows from a write handle to an object on the next operation, if `is_flow_mediated` is set; otherwise, it flows immediately.
- The flow of `current_min_lbl` is captured by the invariants 14 through 17. Whenever any of the quantities involved in these invariants change, `current_min_lbl` needs to be recomputed so as to ensure that those invariants hold. For performance reasons, these recomputations should be done incrementally. In particular, the common cases where the old and new value are the same should be recognized and handled efficiently.
- Unlike `current_lbl`, which is propagated across read-handles only when reads take place, increases to `current_min_lbl` are propagated immediately. Moreover, there is a possibility that the propagation may fail. So, we need to define a function `can_inc_current_min_lbl(X, l)` that returns true iff `current_min_lbl` of entity `X` can be increased to `l`. (If `l` is less than `current_min_lbl(X)` then it returns true.)
- There are many operations that require changes to `current_min_lbl`, such as opening a file for writing, explicit increases to `discr_invul_lbl`, etc. Before such operations can

succeed, `can_inc_current_min_lbl()` needs to be checked. This requires a two-phase protocol, where the first phase is a call to `can_inc_current_min_lbl()`. Note that a call of this function on a subject may result in a recursive call of the same function on its read handles, which in turn may call `can_inc_current_min_lbl()` on objects or other subjects and so on. When this first phase succeeds, then the updates effecting the original operation (write open, increase to `discr_min_lbl`, etc.) need to be performed in the second phase. The second phase will also update the cached value of `current_min_lbl` associated with the entities involved in the first phase.

Obviously, care needs to be taken in implementing this two phase protocol to avoid race conditions, loops, or deadlocks. Performance is always a consideration in the sense that most opens should not require additional locks.

- Handles are created by open operations and destroyed by close operations. (Note that we are using "open" and "close" abstractly here – many concrete OS operations will map to these abstract operations. Indeed, some OS-operations may map to multiple open or close operations, e.g., a socket connection, which, in our model, requires 4 handles to be created.)

  A new write-open operation may require increasing `current_min_lbl` of subjects and objects involved, but if this increase is not possible, then the open will be denied. Similarly, when a read-open operation is performed, the input object must be capable of supporting the integrity level required by the subject, or else the operation will be denied.

- When handles are destroyed, `current_min_lbl` values associated with entities involved may change. Note that the creation/destruction of a handle will not only affect the subject performing the open or close, but other subjects and objects that have an information flow relationship with it. Thus, a single open (or close) may require `current_min_lbl` recomputation for many (potentially all) of the entities on the OS. Hence it is important that these values need be recomputed incrementally — again, the most important requirement is to handle the common cases efficiently.

- Note that when an object or subject is destroyed, the associated handles are also destroyed.

- Hooks for some security events may be missing, and we need to develop methods to enforce our policies in spite of these misses. For instance, there is no LSM hook for close operation. To deal with this, we need to maintain additional information in handles that allows us to check if a certain handle is still valid at runtime, or if it has been closed.

  – Before denying any read access (or read open) due to violation of `current_min_lbl` requirement, we need to verify that `current_min_lbl` on the subject is correct (and is not too high because of failure to recognize that a certain write-handle has been closed).
  – Before denying any write access (or write open) due to violation of `current_min_lbl` requirement, we need to verify that `current_min_lbl` on the object is correct (and is not too high because of failure to recognize that a certain read-handle has been closed).

A.2 Mapping PPI Abstract Operations to LSM hooks

A.2.1 Mapping Subject Operations to LSM Hooks

PPI policy enforcement is implemented using LSM hooks. Much of the work performed falls into two categories:

- Updating the data structures described in the Section A.1 in response to various operations on subjects and objects; and maintaining the invariants listed above after each such operation,
- Storing information that is available in one LSM hook so that it can be used in a subsequent hook where it is needed; and more generally, reconstructing information needed by PPI that is not directly available in the LSM hooks.

Below, we address the last task first, and then proceed to describe PPI actions taken in response to various operations on objects and subjects.

Process creation: `fork` and `clone` operations

Figure A.1 depicts the sequence of invocation of `fork` and `clone` related hooks. A new `PPI_Subject` is created when a new process is created. To identify if it is a fork or clone event, PPI will use a flag `clone_flags` which is available in the hook `security_task_create`. The list of relevant hooks and the operations performed within these hooks are described below. We use the convention that for an LSM hook named `security_X`, the corresponding call back function in PPI is called `ppi_X`.

`security_task_create`: This hook is invoked when a new process is created. PPI will use this hook to distinguish a fork from a clone event, and remember it.

Details

```
if clone_flags is set to 18874385 then set
     current->sub_sec->is_same_subject_group to 0
     /* clone_flags = 18874385 is an identification of "Fork" event */
else set current->sub_sec->is_same_subject_group = 1
```

`security_task_alloc`: This hook is invoked to allocate security structure to a new process being created. In the case of a clone operation, all that is needed is to create a new `PPI_subject`, and set its subject group to point to that of the parent. If it is a fork operation, then a different set of actions are required. Specifically, a new `PPI_subject_group` is created; a new `PPI_subject` is created for the child process, and its subject group field is initialized to point to this new subject group. In addition, the child process gets its own copies of handles of the parent. All these copy operations, naturally, will need to preserve all the invariants noted in Section A.1.4 — for instance, the newly created subject group's list of read and write handles must be exactly the set of child's copies of read and write handles possessed by the parent.

Process Execution

Figure A.2 depicts the main LSM hooks related to `execve`.
Only `ppi_bprm_check_security` and `ppi_bprm_apply_creds` are sufficient to ensure complete mediation because they are the last 2 hooks to be invoked in the context of process execution. A non zero error code returned from `ppi_bprm_check_security` undoes the

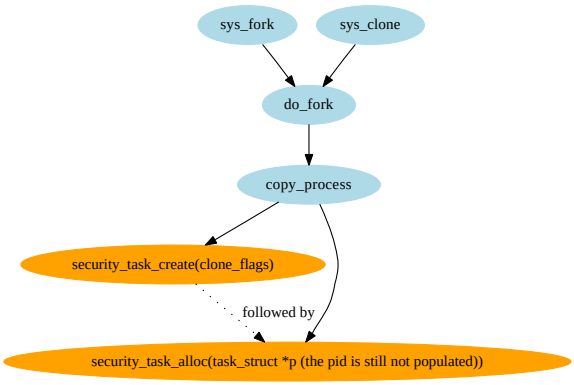

Figure: Task Fork and Clone

Figure A.1: LSM hooks related to `fork` and `clone`

operations performed, from the begining of task execution up-till that point. Hence all checks can be delayed untill `ppi_bprm_check_security`.

`ppi_bprm_check_security`: This hook is invoked when a new program is `execve`'d. The following steps need to be taken at this point:

- An `execve` operation terminates all threads except the one that has made this call. PPI thus deletes all `PPI_subject` structures corresponding to those threads. (There does not seem to be a need to create a new `PPI_subject_group`.)
- For all handles of the subject, set `is_flow_mediated`. (According to `execve` documentation, shared memory segments are closed and mmaps are not preserved.)
- Update the subject label `S` based on the label `F` of the file being `execve`'d
  - `S.current_lbl = min(S.current_lbl, F.subj_lbl.current_lbl, F. obj_lbl.current_lbl)`
  - `S.X = min(S.X, F.subj_lbl.X)`, for `X` in {`is_super`}
  - `S.X = max(S.X, F.subj_lbl.X)`, for `X` in {`read_log`, `write_log`, `min_lbl`, `discr_min_lbl`}
  - `S.X = F.subj_lbl.X`, for all other fields.

  Notes.
- Other fields of the subject label, or the labels of the objects/subjects communicating with this subject, may need to be changed in order to maintain the previously stated data structure invariants.
- `execve` fails if the above operations cannot be done, e.g., if `min_lbl` cannot be increased to the required value. To deal with this possibility, we make a copy of the subject and subject group before making updates, and then rolling back to those copies when failure occurs.
- We may need a mechanism for executing a child process at a higher level of integrity than

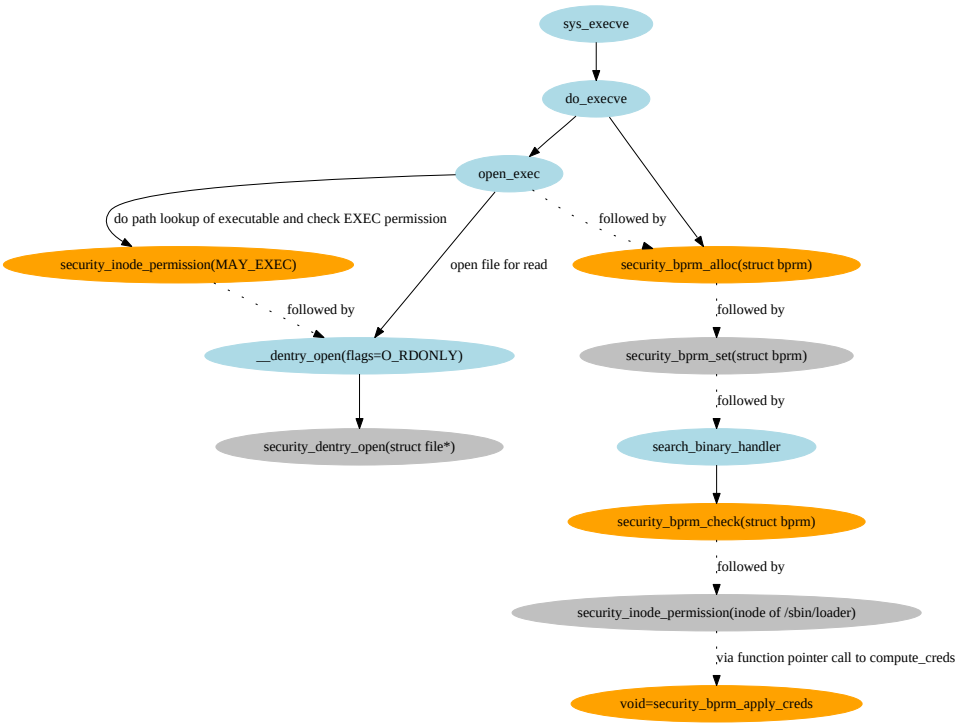

Figure: Task Exec

Figure A.2: LSM hooks related to `exec`

the subject. This is similar to setuid processes. As in the case of setuid processes, lots of checks need to be made regarding who controls inputs (or can send signals to) higher integrity children. This may be done as part of future work.

- We do not bother with close-on-exec fds. The kernel calls the `security_file_free` hook for each descriptor closed in this fashion, when the reference count of `struct file` drops to 0. However there is no hook for mediating `fput` calls, in other words, PPI can not detect file close events which do not make the reference count 0. Consequently PPI is unable to close a subject's PPI handles on an object that it may have closed. To tackle this, PPI treats every handle as "tentative," and before each use, verifies that the handle is still valid. Invalid handles are promptly deleted, and removed from the attributes maintained with subjects and objects.

`ppi_bprm_apply_creds`: This is the last hook invoked in the sequence of hooks invoked for task execution. It does not return any error code. However the PPI related subject attributes are updated in this hook.

Process Exit

The LSM hook `ppi_task_free` is invoked when the process exits. The following actions are taken:

- Free the `PPI_subject` data structure for the exiting process.
- Decrement the ref count field for the `PPI_subject_group`. If it becomes zero, the subject group is freed. This will in turn mean that all the handles associated with the subject group will be freed. (On process exit all the open file descriptors will be closed by `file_free_security` hook, so the list of handles will likely be empty in most of the cases.)

A.2.2 Mapping Object Operations to LSM hooks

We first address the problem of correlating objects to their handles, and then proceed to describe the details of LSM hook mappings and the necessary checks for enforcement.

Mapping objects to handles and vice-versa

LSM hooks related to input/output operations typically provide a pointer to a file structure (specifically, `struct file *`) maintained by the kernel, whereas our enforcement actions require us to identify corresponding handles, which are purely local to PPI. To perform this association efficiently, PPI maintains two arrays `fd2rhandle` and `fd2whandle` that are indexed by a file descriptor and yields a pointer to the corresponding handle. Unfortunately, this is not enough — as noted before, for a variety of reasons, fd information may be stale or inaccurate; and handles can be stale. To cope with these problems, we rely on the following helper functions:

- `update_handle_fd(PPI_handle *h, PPI_subject *s, int fd, bool handle_type)`:
  This function is called when we detect that the fd associated with a handle has changed to a new value.

Algorithm

```
1  /* For brevity, we only show the case where handle_type is ''read''*/
2  h->fd = fd;
3  s->group->fd2rhandle[fd] = h;
```

- `validate_handle(PPI_handle *h, PPI_subject *s)`: This function is called to check if a handle is current, and if so, to get the correct fd associated the handle. If the fd stored within the handle is correct, then this function is very fast; otherwise, it requires a search through the file descriptor table of a process, which takes a bit more time. If this search does not find the handle, then the handle must already have been closed, and so we clean up the handle.

Algorithm

```
1  struct file *f = fget(h->fd);
2  if ((f != NULL) && inode_equal(h, f) && mode_compatible(h, f))
3     return h->fd;
4  /* The fd-table is maintained by the kernel for all processes */
5  for each file struct f in the fd-table of s do
6     if (inode_equal(h, f) && mode_compatible(h, f)) then
7        update_handle_fd(h, s, getfd(f), handle_type(h))
8        return getfd(f)
```

```
9    done
10   /* Subject must have closed the corresponding fd. Free the handle, update current˙min˙lbl, etc. */
11   free_handle(h, s)
```

- `get_handle(PPI_subject *s, struct file *f, bool handle_type)`:

Algorithm

```
1    fd = ppi_getfd(f); /* lookup the fdtable to find the corresponding index */
2    h = s->group->fd2rhandle[fd];
3    if (h != NULL) then
4       if (inode_equal(h, f) && mode_compatible(h, f))
5          then return h;
6       else validate_handle(h, s);
7    /* The handle's fd is not current. Use sequential search to find the right handle. */
8    for each handle h in s->group->read_handles do
9       if (inode_equal(h, f)) then /* Update fd information */
10         update_handle_fd(h, s, fd)
11         return h;
12   done
13   /* Should never reach here; print error message */
```

If a valid, fresh handle is found at any point during the above search, then we stop the search at that point. By not continuing the search, there is a possibility that we leave stale handles in the `PPI_subject` data structure, but this is acceptable since such handles will be detected before they are used.

The only advantage of maintaining the `fd2handle` array is that in the typical case, we will be able to identify the handle after just a single array lookup. In the event that a full search is needed, then the identified handle is inserted into the `fd2handle` arrays at the location given by their current fd. This ensures that subsequent lookups of this handle will take constant time until the fd is involved in another `dup`-like operation.

File Operations

File creation. LSM hooks relevant to creation of file objects is shown in Figure A.3. PPI implementation makes use of the following hooks in this regard:

`ppi_inode_create`: This hook is invoked for regular files and is used for performing the `ppi_bind` abstract operation on the inode of the parent of the inode being created.

`ppi_inode_alloc`: This hook is invoked to allocate in-memory security structure to a new inode being created. PPI will use it to assign integrity label to inode. This hook primarily invokes the `ppi_create` abstract operation, which then does the object's security allocation and initialisation of its label.

`ppi_inode_init`: This hook is invoked to store security information with the inode on disk. PPI will use it to store in-memory label onto disk to make it persistent.

File deletion. LSM hooks relevant for file deletion are shown in Figure A.4. Specifically, the following hooks are used:

`ppi_inode_unlink`: This hook is invoked when an inode is being deleted from disk. If a temporary PPI write handle can be created successfully then only unlink hook will succeed. The `ppi_unlink` abstract operation is called from this hook.

`ppi_inode_free`: This hook is invoked in following two cases:

- deletion of an in-memory inode from cache when there are no more references to that inode.
- deletion of an inode resident on persistent store

In all cases, there are no more references to the inode, and so PPI can deallocate data structures allocated to store information related to the object. This hook invokes the `ppi_clean_object` abstract operation.

File open. LSM hooks related to file opens are shown in Figure A.3.

`ppi_inode_permission`: This hook is invoked to check the permissions on the inode before performing any operations on it. The type of operation is available through a parameter called `mask`.

Current label will be propagated from a read handle to its subject only when the first bytes of data are actually read from the file. When object is opened in read-write mode, read handle will be created first, and then the write-handle. This hook invokes the `ppi_open` abstract operation.

File close. Figure A.4 illustrates the hooks relevant to this operation. In particular, we use the hook `ppi_file_free`. At this point, the handle is freed, and `current_min_lbl` of the associated objects and subjects is adjusted. (This adjustment can decrease `current_min_lbl` or leave it the same, but not increase it.) The handle is also purged from the list of handles maintained by the object and the subject.

Unfortunately, this hook is invoked only when the last file descriptor (across all processes) for a file is closed. This means that for files opened by multiple processes, all but the last close operation will be invisible to PPI. To cope with this, we use the helper function `validate_handle` (described earlier) before every use of a handle.

File read and write. The relevant hooks are shown in Figure A.5.

`ppi_file_permission`: This hook is invoked to check permissions on a file before it is read/written. Note that only basic enforcement rules need to be checked here: constraint propagation is done at the time of open operation. This hook invokes `ppi_read` and `ppi_write` abstract operations. Recall that all handles are validated before they are used.

File mmap. The relevant hook is discussed below.

`ppi_file_mmap`: The file needs to be opened before it can be mmaped. Therefore, the handles must already have been created before this hook is reached. This hook has two important input parameters: a file structure and flags. Flags describe what accesses are permitted to the data being mmaped. PPI will use this hook to update the file descriptor information associated with the handle. Finally, the handle's `is_flow_mediated` is set to false. Flags will be used to identify if the read handle or write handle or both are involved.

Directory operations

The way PPI handles directories is better explained by understanding the operations that are performed on a directory.

- Directory Creation : This operation is no different than creating a regular file. The usual `ppi_bind` will be called on the parent directory from the `ppi_inode_mkdir` hook.
- Directory Renaming : Since a rename operation is on an inode we do not distinguish between regular files and directories. The enforcement in `ppi_inode_rename` applies to directories as well.
- Directory Removal : This operation is just like the unlink operation on regular files and hence we simply invole `ppi_unlink` abstract operation from `ppi_inode_rmdir` hook.
- File Creation in a Directory : This operation is covered under the file creation operations discussed earlier.
- Directory Traversal : This operation mainly deals with reading the contents of a directory or the pathname resolution of the directory. In either case the event can be mediated in `ppi_inode_permission`. However we currently do not invoke the `ppi_lookup` abstract operation from this hook for reasons discussed in Section 2.2.2

Hard Links

As discussed in Section 2.2.1, there is no provision in the current LSM framework to associate labels with hard links. Hence the only check that is made in the current implementation is that of link creation, which is in-line with most object creation operations. We simply invoke the `ppi_bind` abstract operation in `ppi_inode_link` hook.

Symbolic Links

Symbolics links have been implemented in our framework, completely as per the discussion in Section 2.2.1 and Section 2.2.2. We simply invoke the `ppi_bind` abstract operation in `ppi_inode_symlink` hook, which pertains to symlink creation. Also, we update the value of subject's `virtual_current_label` in `ppi_inode_follow_link`. This operation is the key to downgrading a subject if it tries to open an object via a low integrity link.

File systems

As discussed in Section 2.2.1, a mount operation is treated as a combination of a removal of the mount point directory, a write to the device, and a bind to the mount point directory. Important file system operations are as follow:

- File system mount : The hook involved for this operation is `ppi_sb_mount`. The 2 abstract operations that map to this hook are `ppi_bind` and `ppi_delete`. In particular `ppi_delete` operation needs to be called once for the mount point and once for the device that represents the file system.
- File system unmount : The hook involved for this operation is `ppi_sb_umount`. The ab-

stract operations mapped to this hook are exactly the same as those for the mount operation.

Named Pipes

Pipes provide one of the most basic inter-process communication mechanisms. Pipes are of two types, named and unnamed. Unnamed pipe is a in-memory kernel data structure that is referenced using file descriptors, but there is no way to externally name these data structures. Named pipes, in contrast, are associated with a name in the file system, and hence can be referenced externally. In addition, permissions can be associated with these file names that can control who can connect to a pipe. As a result, named pipes are opened and read/written in much the same was as files, whereas unnamed pipes do not have explicit open operations.

Named pipe creation. Figure A.6 depicts the hooks involved in the creation of named pipes. In particular, we use some of the same hooks as for creation/opening of regular files.

`ppi_inode_mknod`: This hook is invoked to create named objects only. To distinguish between the named and unnamed pipes, a flag `is_named_pipe` is maintained in the subject security structure.

`ppi_inode_alloc, ppi_inode_init`: These two hooks are handled in the same way as files.

Named pipe open, close, deletion. All the operations on named pipe are the same as regular files from the point of view of policy enforcement in our framework.

Unnamed Pipes

Figure A.7 depicts the hooks involved in the creation of unnamed pipes.

Object creation. The hooks for this operation are enlisted below.
`ppi_inode_alloc`: This hook simply invokes the `ppi_create` abstract operation.

`ppi_d_instantiate`: As mentioned earlier, unnamed pipes are implicitly opened on creation. To achieve this effect, PPI will create two handles (one for the read-pipe and another for write-pipe) in this hook. Note that PPI can distinguish between named and unnamed pipe creations by examining the field `is_named_pipe`.

Internet Stream Sockets

The various operations on Internet Stream Sockets have been discussed below.

Socket Creation. LSM hooks relevant to creation of socket objects is shown in Figure A.8. PPI implementation makes use of the following hooks at the time of socket creation (the only exception is the creation of the new 'accept' socket, which is not captured at the server side)

`ppi_inode_alloc` : As discussed earlier this hook is invoked for creation of an inode that is referred to by the socket. The abstract operation `ppi_create` is called in this hook.

`ppi_sk_alloc_security` : This hook is responsible for creation of the read and write handles on the socket. A socket always has read and write sockets associated with it and therefore both these must be created at the time of socket creation. Abstract operation `ppi_open` maps to

this hook and needs to be invoked twice to create read and write handles.

Socket Close. This operation simply closes the read and write handles established on the socket. The work is done by `ppi_close` abstract operation in `ppi_file_free_security`. Invocation of `ppi_file_free_security` indicates that the reference count of the `file` structure associated to the file is zero and there are no users of that socket. The write handle(s) on the closing sock may belong to other subject(s) which means that at this point those all the write handle(s) too will be closed.

Socket Listen. No PPI checks are required on socket listen.

Socket Connect. Section 2.2.2 gives the description of `ppi_connect` abstract operation. The implementation of socket connect operation matches that description exactly. However no changes made at the time of `ppi_connect` abstract operations are committed, otherwise it could be extremely easy for a low integrity process to downgrade a high integrity server by simply calling a `connect` system call, even if the connection is denied by the server at the time of `accept`. The hook for this operation is enlisted below and the flow of the relevant kernel hooks is shown in Figure A.9

`ppi_socket_connect` : Abstract operation `ppi_connect` will be called from this hook. Since this call is made by the client, the client socket information is available. The information about the server's socket (at this point the server's socket refers to the server's listening socket because the new socket is created at the time of *sys_accept*) can always be obtained by a lookup in the server process's file descriptor table, if the server is running on the same local machine.

Socket Accept. No enforcement is done for this operation because the client information is not available in `ppi_socket_accept`. The flow of the relevant kernel hooks, for this operation, is shown in Figure A.10.

Socket Send/Receive. TCP is a connection oriented protocol. Therefore once the connection is established, before any communication starts on the TCP connection, we must associate the write handles with the peer objects. Since we do not have any hook with error return capability after the connection is established, we enforce integrity policies in `ppi_socket_sendmsg` hook. On first write, it will be checked if the association can be formed. Note that there is no enforcement done for `ppi_socket_recvmsg` because there is no way to know the source of the message till the data has been read. So we rely completely on sender side checks. The hook for this operation is indicated below.

`ppi_socket_sendmsg` : The abstract operation `ppi_sendto` maps to this hook. The abstract operation is a combination of `ppi_connect` and `ppi_write`.

Internet Datagram Sockets

UDP is a connectionless protocol. Therefore the association will be established only when client tries to send to server and vice-versa. The association will propagate `current_min_lbl` and `current_lbl`. Once the propagation is done the association with the peer socket will be broken. Note that we cannot enforce in `ppi_socket_recvmsg` hook since in case of UDP we do not have the receiver information until we actually receive the data. PPI operations for Datagram

Sockets are done in the following hook:

`ppi_socket_sendmsg` : The abstract operation `ppi_sendto` maps to this hook. The abstract operation is a combination of `ppi_connect` and `ppi_write`.

UNIX domain sockets

UNIX domain sockets are much like Internet sockets except that communications are between processes on the same local machine and the data transfer does not involve kernel networking protocol stack. The data transfers does not use the file system. Rather this mechanism uses kernel memory buffers for the actual transfers. In Unix domain sockets, the socket address structure consists of a field `sun_path` which can be either null-terminated file system pathname or an abstract name. The discussion for UNIX domain sockets is divided into 2 parts:

- UNIX domain stream sockets
- UNIX domain datagram sockets

UNIX domain stream sockets

In case of TCP with socket having file system pathname two sockets are created by client and server processes using `socket` system call. The resulting objects having file system pathname. For such a socket that has a file system pathname, our framework creates a special handle called the *create-write handle*.

A note on create-write handles. In case of UNIX domain sockets, the socket can be created on the file system with file system pathname. Even though according to unix access control, no one can read from or write into the socket, this socket file can be moved, overwritten. Therefore it is important to protect the integrity of the socket file. To to do this we introduce a new type of PPI handle other than read and write handles. This handle is known as create-write handle. This type of handle will be created by the subject that creates the object. Note that such handle will be created only for on disk-objects. (For in-memory objects like socket, unnamed pipe, their object creation always results into immediate read/write handle creation.) The create event indicates that creating subject intends to use it or make it available for use to other subjects. When a new object is created, `ppi_inode_init_security` hook (this hook is invoked immediately after `ppi_inode_alloc_security`), creates a create-write handle for the object. If the inode creation fails then `ppi_inode_free_security` will clean up the create-write handle from the system. It is important to note that the handle validation procedure will not be applied for create-write handles.

This handle will be of write type since it is a representation of the subject that created it. Creating such handle will ensure that the integrity changes to a subject are also propagated to its newly created objects. This propagation also ensures that a subject itself does not lose control of its newly created object. Create-write handles will be removed from the system in one of the following two ways:
1. The object corresponding to the handle is deleted.
2. The subject which created the handle gets terminated.

Domain Socket Creation. LSM hooks relevant to creation of socket objects is shown in Figure A.8. PPI implementation makes use of the following hooks at the time of socket creation (the only exception is the creation of the new 'accept' socket, which is not captured at the server side)

`ppi_inode_alloc` : This hook helps in for socket object creation. The abstract operation `ppi_create` is called in this hook.

`ppi_sk_alloc_security` : This hook is responsible for creation of the read and write handles on the socket. Abstract operation `ppi_open` maps to this hook and needs to be invoked twice to create read and write handles.

Domain Socket Close. This operation simply closes the read and write handles established on the socket. The work is done by `ppi_close` operation in `ppi_file_free_security`.

Domain Socket Connect. The implementation for this operation is exactly like the implementation for `connect` operation for Internet stream sockets. Even though integrity checks are made in this hook, no associations are committed. The reason is that for permanent associations, the peer object is unavailable (has not been allocated). The peer object is created only at the time of `accept`. The hook for this operation is enlisted below and the flow of the relevant kernel hooks is shown in Figure A.9

`ppi_unix_stream_connect` : Abstract operation `ppi_connect` will be called from this hook.

Domain Socket Accept. No enforcement is done for this operation because the client information is not available in `ppi_socket_accept`. The flow of the relevant kernel hooks, for this operation, is shown in Figure A.10.

Domain Socket Send/Receive. No enforcement is done for `ppi_socket_recvmsg` because there is no way to know the source of the message till the data has been read. So we rely completely on sender side checks. The hook for this operation is indicated below.

`ppi_socket_sendmsg` : The abstract operation `ppi_sendto` maps to this hook. The abstract operation is a combination of `ppi_connect` and `ppi_write`.

UNIX domain datagram sockets

The only enforcement done for UNIX domain datagram sockets, in our framework, is at the time of sending data from one peer to another. The description of operation and details of implementation are same as that for the Internet datagram sockets. The hook used for enforcing policies at the time of data transmission is indicated below.

`ppi_unix_may_send` : The abstract operation `ppi_sendto` maps to this hook.

System V Inter-process Communications (IPCs) : Shared Memory

PPI framework operations for shared memory have been discussed below. `kern_ipc_perm` is the kernel data structure that represents the System V IPC objects in the system and is the common data structure for all IPCs.

Shared Memory Object Creation. Whenever a new shared memory IPC is created, kernel creates a new data structure `kern_ipc_perm`. It contains a security field in which an object security information can be stored. In case of a shared memory, the data structure is `shmid_kernel`. This contains the pointer to `kern_ipc_perm` and a pointer to `struct file` (`shm_file`). PPI object will also contain a pointer to `shm_file` for future shared memory handle validation. The hooks are described below.

`ppi_shm_alloc_security`: This hook is invoked whenever a new shared memory segment is created. PPI will associate security information with the shared memory object in this hook.

`ppi_shm_shmat`: Handle creation occurs in this hook. The idea is same as for handle creation for files in `ppi_inode_permission`. This hook specifies the intent. If intent is read-only then read handle will be created. Otherwise, both read and write handles will be created. The flag `is_flow_mediated` is set to false.

Shared Memory Handle Destruction. There is no hook for tracking shared memory detach event. The system call `sys_shmdt` simply unmaps the shared memory. Therefore shared memory handles will be validated before they are used to make sure that stale PPI handle do not remain in the system.

Shared Memory Handle Validation. Handle validation for shared memory may have to be done differently. Every process has its own virtual memory mapping. Every `vm_area_struct` of the process has a file pointer `vm_file`. Whenever process attaches to the shared memory it creates new `vm_area_struct` whose file pointer `vm_file` points to `shm_file` in the global shared memory object. If the process is truly attached then this file pointer should match otherwise it is an invalid handle.

`ppi_shm_free_security`: This hook is invoked when the reference count on shared memory falls to zero. The clean-up of all handles and the shared memory object occurs in this hook.

Figure A.3: LSM hooks related to opening of regular file

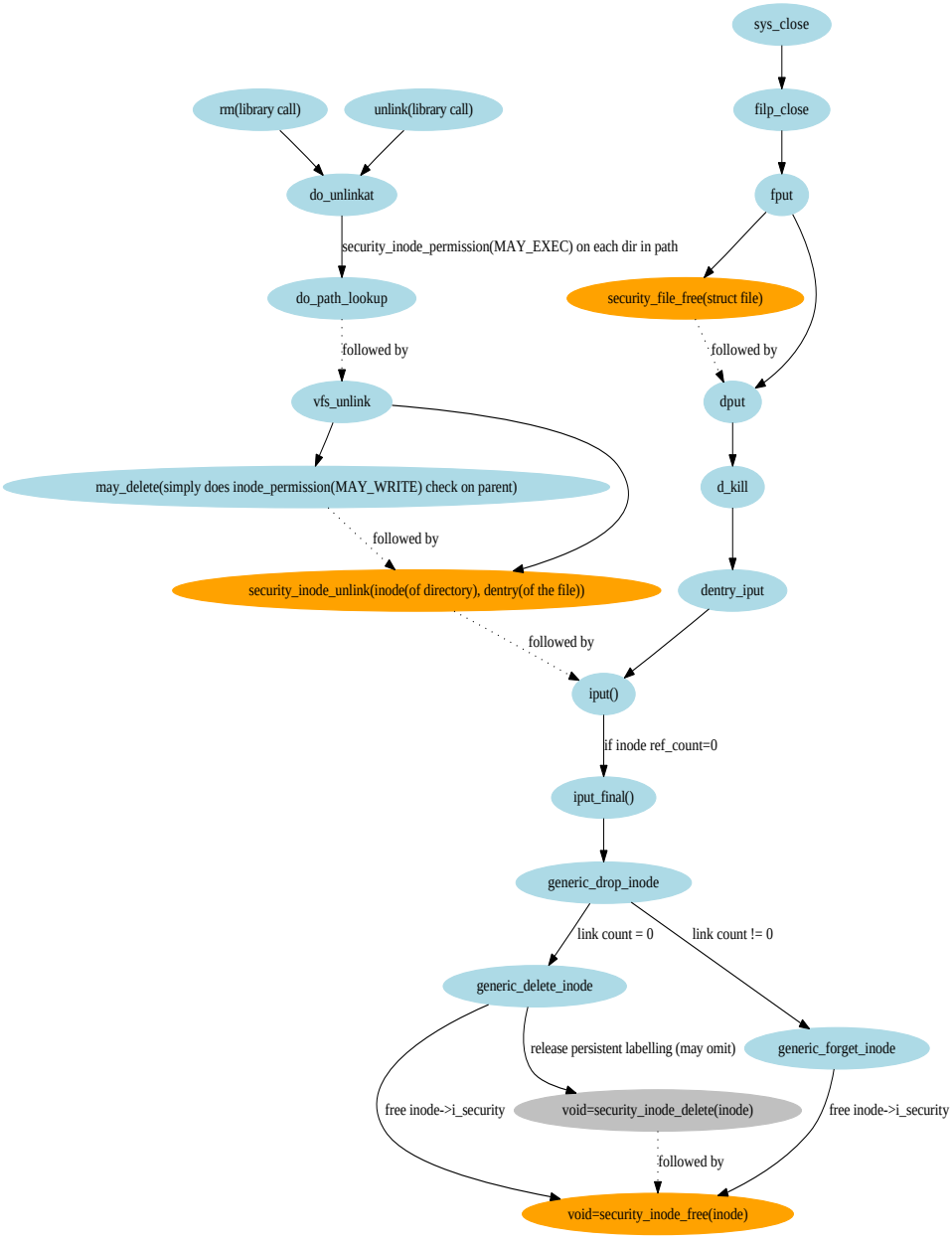

Figure A.4: LSM hooks related to deletion of regular files

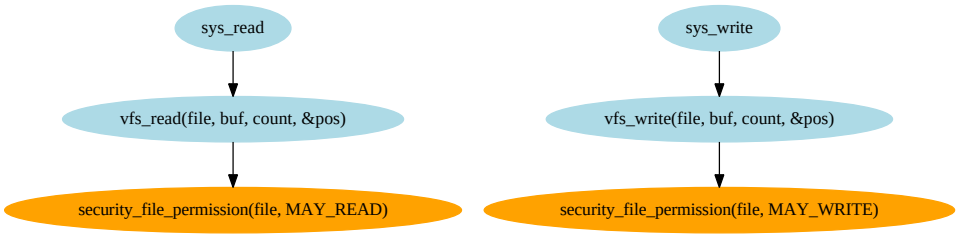

Figure: File Read and Write

Figure A.5: LSM hooks related to reads and writes of regular files

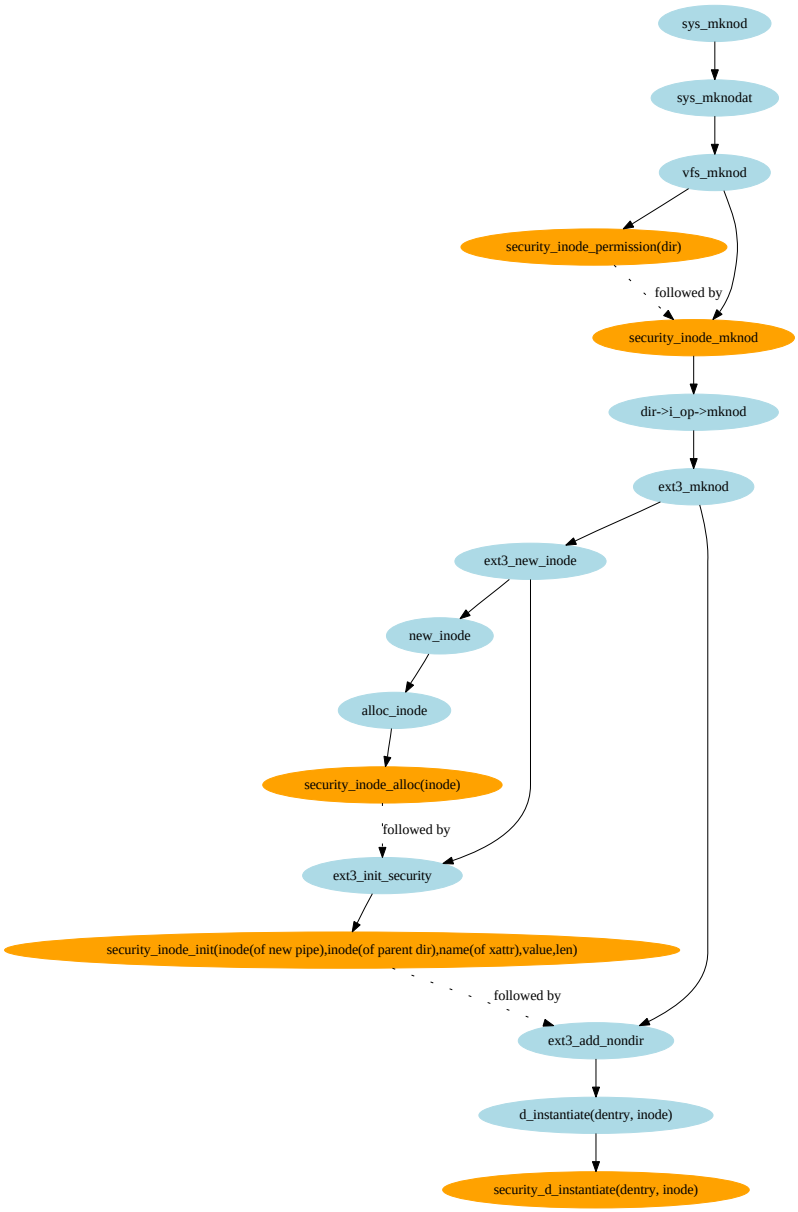

Figure: Named pipe create

Figure A.6: LSM hooks related to creation of named pipes

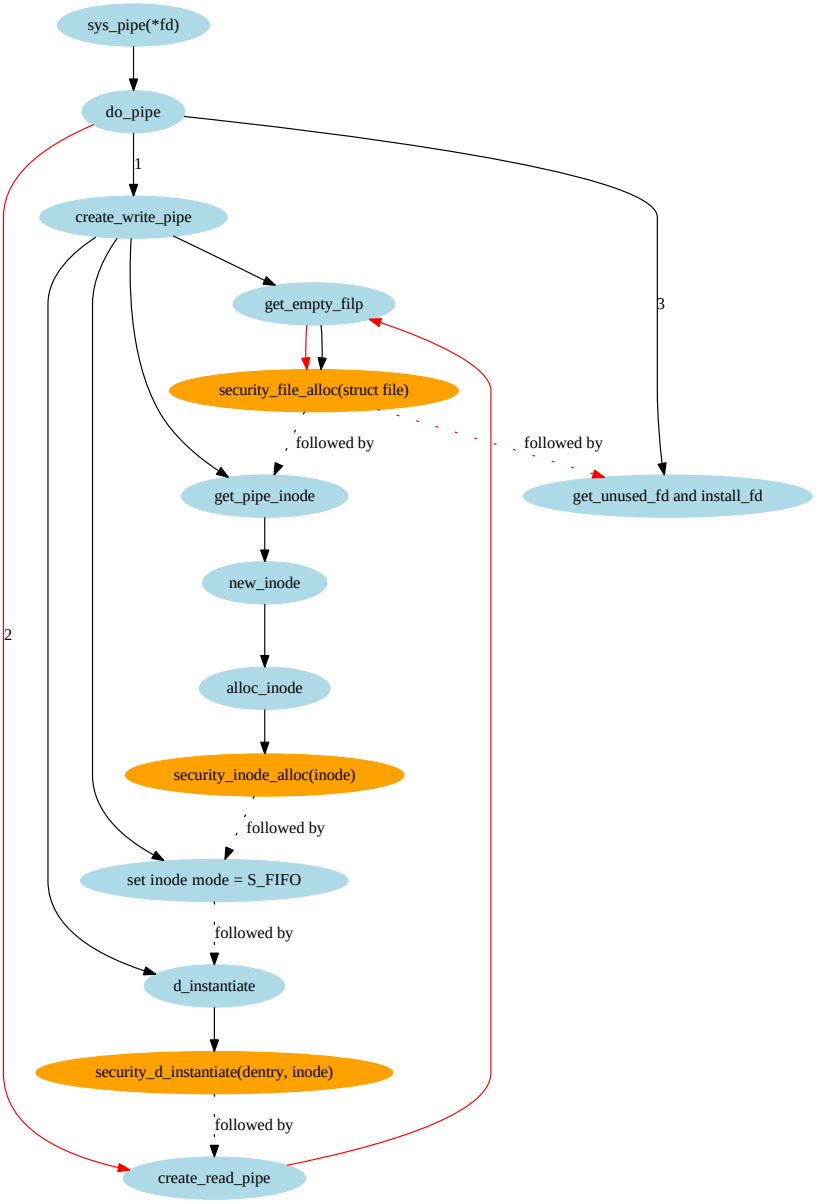

Figure: Un-named pipe create and open

Figure A.7: LSM hooks related to creation of unnamed pipes

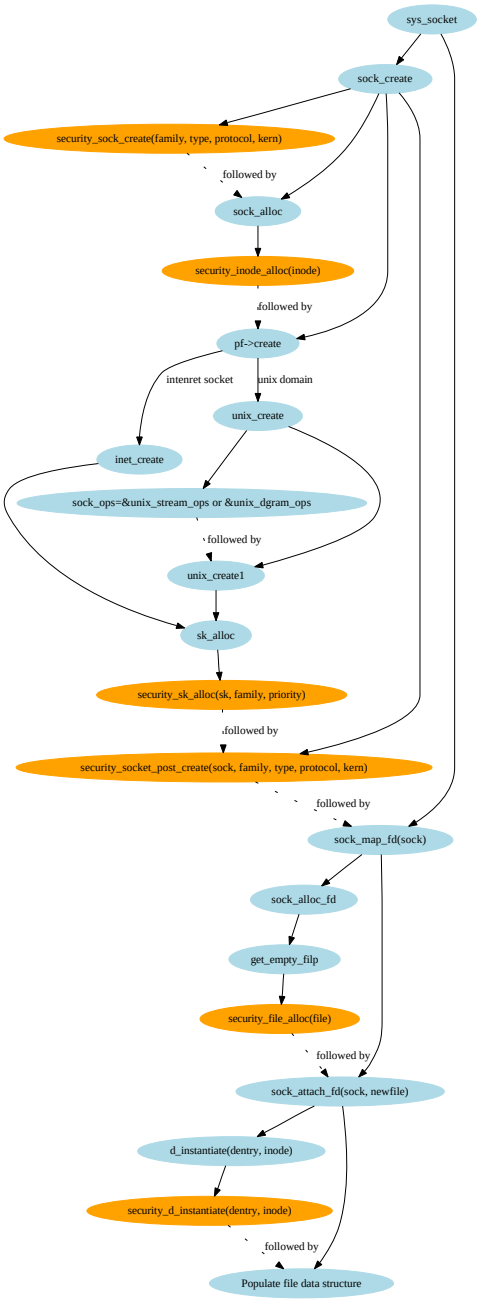

Figure: socket create

Figure A.8: LSM hooks related to socket creation

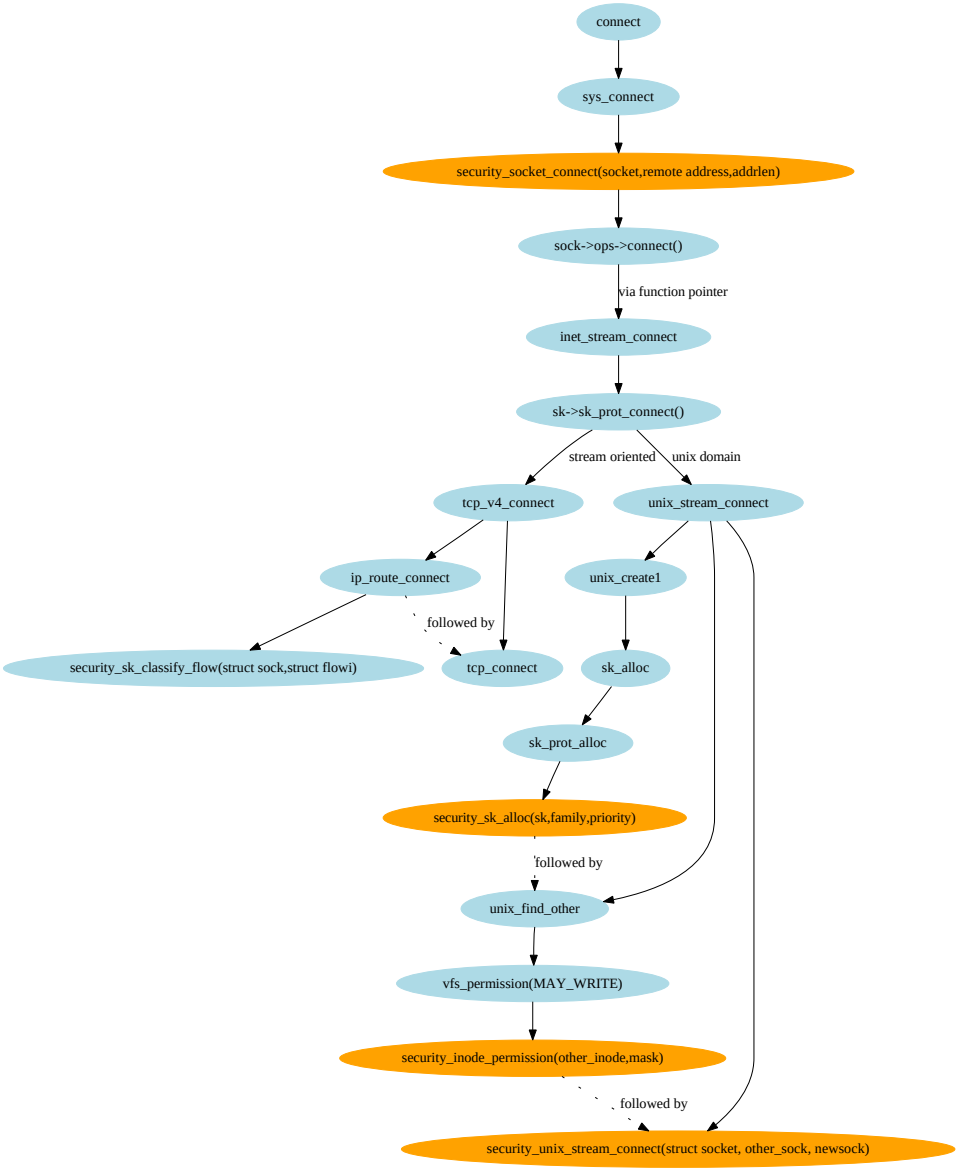

Figure: Socket Connect

Figure A.9: LSM hooks related to socket connect

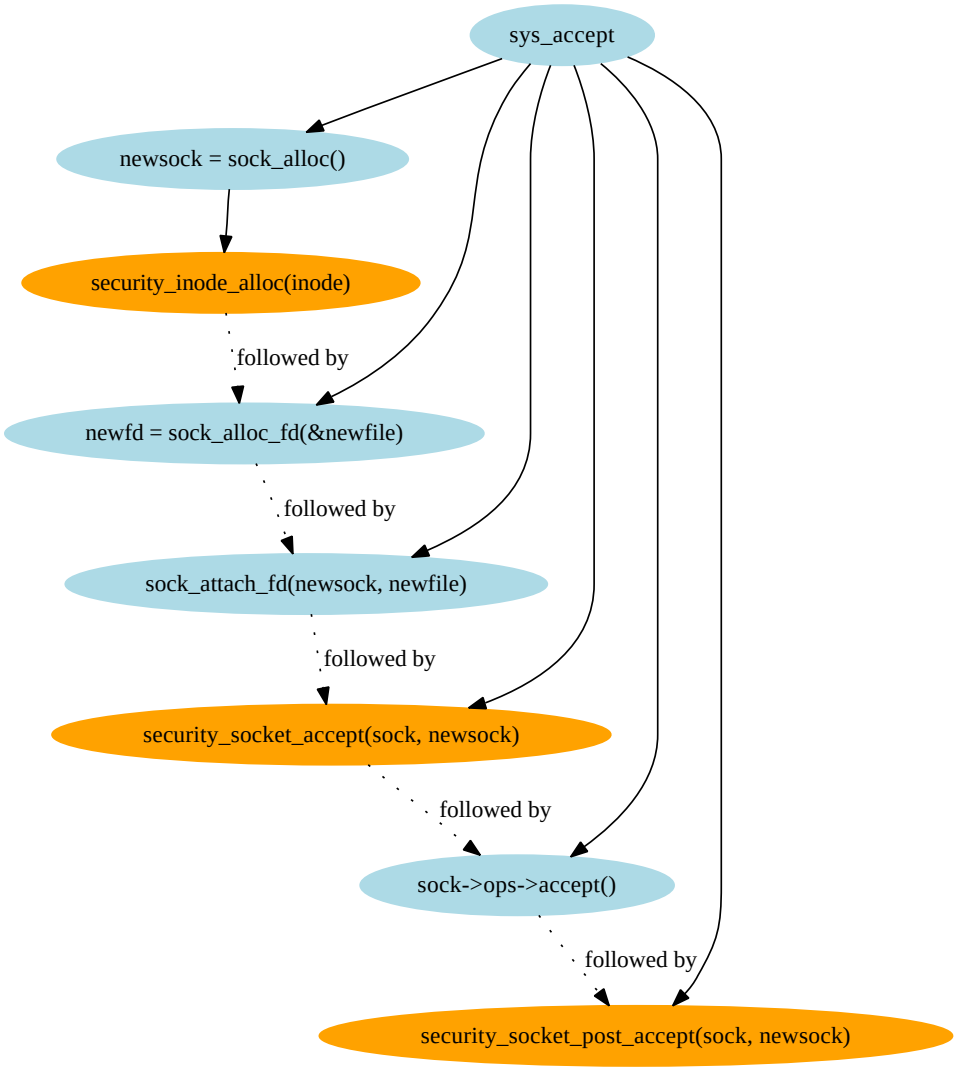

Figure: socket accept

Figure A.10: LSM hooks related to socket accept

Bibliography

[1] Anurag Acharya and Mandar Raje. Mapbox: Using parameterized behavior classes to confine applications. In *USENIX Security Symposium*, 2000.

[2] Alcatraz. http://www.seclab.cs.sunysb.edu/alcatraz.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, volume 37, 5 of *Operating Systems Review*, pages 164–177, New York, October 19–22 2003.

[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] Timothy Fraser. Lomac: Low water-mark integrity protection for COTS environments. In *IEEE Symposium on Security and Privacy*, 2000.

[6] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *USENIX Security Symposium*, 1996.

[7] Francis Hsu, Thomas Ristenpart, and Hao Chen. Back to the future: A framework for automatic malware removal and system repair. In *Annual Computer Security Applications Conference (ACSAC)*, December 2006.

[8] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. Technical report, National Security Agency, February 2001. `www.nsa.gov/selinux/papers/slinux.pdf`.

[9] M. Douglas McIlroy and James A. Reeds. Multilevel security in the UNIX tradition. *Software - Practice and Experience*, 22(8):673–694, 1992.

[10] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, Washington, DC, August 2003.

[11] D. Safford and M. Zohar. A trusted linux client (tlc). In *IBM Research*, 2005.

[12] W. Sun, Z. Liang, R. Sekar, and VN Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[13] Weiqing Sun, Zhenkai Liang, R. Sekar, and VN Venkatakrishnan. One-way isolation: An effective approach for realizing safe execution environments. In *Proceedings of ISOC Network and Distributed Systems Symposium (NDSS)*, 2005.

[14] Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *IEEE Symposium on Security and Privacy*, 2008.

[15] Arjan van de Ven. LSM conversion to static interface. `http://lwn.net/Articles/255666/`, October 2007.

[16] Brian Walters. VMware virtual platform. *j-LINUX-J*, 63, July 1999.

[17] Yang Yu, Fanglu Guo, Susanta Nanda, Lap chung Lam, and Tzi cker Chiueh. A feather-weight virtual machine for windows applications. In *Proceedings of the 2nd ACM/USENIX Conference on Virtual Execution Environments (VEE'06)*, June 2006.