Practical Proactive Integrity Preservation: A Basis for Malware Defense[*]

Weiqing Sun R. Sekar Gaurav Poothia Tejas Karandikar
Department of Computer Science
Stony Brook University, Stony Brook, NY 11794

Abstract

Unlike today's reactive approaches, information flow based approaches can provide positive assurances about overall system integrity, and hence can defend against sophisticated malware. However, there hasn't been much success in applying information flow based techniques to desktop systems running modern COTS operating systems. This is, in part, due to the fact that a strict application of information flow policy can break existing applications and OS services. Another important factor is the difficulty of policy development, which requires us to specify integrity labels for hundreds of thousands of objects on the system. This paper develops a new approach for proactive integrity protection that overcomes these challenges by decoupling integrity labels from access policies. We then develop an analysis that can largely automate the generation of integrity labels and policies that preserve the usability of applications in most cases. Evaluation of our prototype implementation on a Linux desktop distribution shows that it does not break or inconvenience the use of most applications, while stopping a variety of sophisticated malware attacks.

1. Introduction

Security threats have escalated rapidly over the past few years. Zero-day attacks have become significant threats, being delivered increasingly through seemingly innocuous means such as web pages and documents. Malware is rampant, being installed on millions of computers around the Internet through implicit or explicit software downloads from untrusted sources. Emergence of cyber crime has led to increasingly stealthy and sophisticated attacks and malware that can hide from the best available defenses today.

Today's malware defenses rely mainly on reactive approaches such as signature-based scanning, behavior monitoring, and file integrity monitoring. Unfortunately, attackers can easily modify the structure and behavior of their malware to evade detection by signature-based or behavior-based techniques. They may also subvert system integrity monitoring tools using rootkit-like techniques. It is therefore necessary to develop proactive techniques that can stop malware before it damages system integrity.

Sandboxing is a commonly deployed proactive defense against untrusted (and hence potentially malicious) software. It restricts the set of resources (such as files) that can be written by an untrusted process, and also limits communication with other processes on the system. However, techniques that regulate write-access without restricting read-access aren't sufficient to address adaptive malware threats. Specifically, they do not satisfactorily address indirect attacks, where a benign application ends up consuming malware outputs stored in persistent storage (e.g., files). For instance, malware may modify the following types of files used by a benign application:

- *System libraries, configuration files or scripts.* One may attempt to eliminate this possibility by preventing untrusted software from storing any files in system directories, but this will preclude the use of many legitimate (untrusted) applications that expect to find their binaries, libraries and configuration files in system directories. Alternatively, one can explicitly enumerate all the files in system directories that are used by benign applications, but this becomes a challenging task when we consider the number of such files — for instance, a typical desktop Linux distribution contains over 100K files in system directories. Errors may creep into such enumerations, e.g., one may leave out optional libraries (e.g., application extensions such as Apache modules, media codecs, etc.) or configuration/customization files, thereby introducing opportunities for indirect attacks.
- *User-specific customization files and scripts.* Identifying all user-specific scripts and customization files is even harder: different applications use different conventions regarding the location of user-specific customization files. Moreover, some of these files may in turn load other user files, or run scripts within user directories. Static identification of all the files used

[*]This research is supported in part by an ONR grant 000140710928 and an NSF grant CNS-0627687.

by a benign application may be very hard.

We observe that significant harm can result from unauthorized modifications to user files. For instance, by altering ssh keys file, malware may enable its author to log into the system on which it is installed. By modifying a file such as `.bashrc`, e.g., by creating an alias for a command such as `sudo`, malware can cause a Trojan program to be run each time `sudo` is used. Worse, malware can first modify a configuration file used by a less conspicuous benign application, such as a word-processor. For instance, it may replace the name of a word-processor plug-in with a Trojan program that in turn modifies `.bashrc`.

- *Data files.* Malware may create data files with malicious content that can exploit vulnerabilities in benign software. The recent spate of vulnerabilities in image and document viewers, web browsers, and word-processors shows that this is indeed a viable approach. Malware may save these files where they are conspicuous (e.g., on the desktop), using names that are likely to grab user attention. When the user invokes a benign viewer on the file, it will be compromised. At this point, malware can achieve its objectives using the privileges available to the viewer.

In contrast, we develop an approach in this paper that aims to *provide positive assurance about overall system integrity.* Our method, called *PPI* (Practical Proactive Integrity protection), identifies a subset of objects (which are typically files) as *integrity-critical* and a set of *untrusted* objects. We assume that system integrity is preserved as long as untrusted objects are prevented from influencing the contents of integrity-critical objects either directly (e.g., by copying of an untrusted object over an integrity-critical object) or indirectly through intermediate files. In other words, there should be no information flow from untrusted objects to integrity-critical objects.

Although information-flow based integrity preservation techniques date as far back as the Biba integrity model [6], these techniques have not had much success in practice due to two main reasons. First, these techniques require every object in the system to be labeled as high-integrity or low-integrity — a cumbersome task, considering the number of files involved (more than 100K on typical Linux systems). Manual labeling is prone to errors that can either damage system integrity (by allowing an integrity-critical file to be influenced by a low-integrity application) or usability (by denying a legitimate operation as a result of security violation). Secondly, the approach is not very flexible, and hence breaks many applications. To overcome this problem, many applications may need to be designated as "trusted," which basically exempts them from the information flow policy. Obviously, an increase in the number of trusted applications translates to a corresponding decrease in assurance about overall integrity.

As a result of the factors mentioned above, information-flow based techniques have not become practical in the context of contemporary operating systems such as Windows and Linux. In contrast, we have been able to develop a practical information-flow based integrity protection for desktop Linux systems by focusing on (a) automating the development of integrity labels and policies, (b) providing a degree of assurance that these labels and policies actually protect system integrity, and (c) developing a flexible framework that can support contemporary applications while minimizing usability problems as well as the need to designate applications as "trusted." Our experiments considered a modern Linux Workstation OS together with numerous benign and untrusted applications, and showed that system usability is preserved by our technique, while thwarting sophisticated malware.

1.1. Goals of Approach

- *Provide positive assurances about system integrity* on a contemporary Workstation, e.g., a Linux CentOS/Ubuntu desktop consisting of hundreds of benign applications and tens of untrusted applications. *Integrity should be preserved even if untrusted programs run with root privileges.*
- *Effectively block rootkits and most other malware.* Most malware, including rootkits and spyware, should be detected when they attempt to install themselves, and removed automatically and cleanly. Stealthier malware should be detected when they attempt to damage system integrity, and can be removed at that point. *We do not address malware that can operate without impacting system integrity,* e.g., a P2P application that transmits user data to a remote site when it is explicitly invoked by a user.
- *Be easy to use,* for end-users as well as system administrators. Usability encompasses the following:
  - *Preserve availability of benign applications,* specifically, provide a certain level of confidence that benign applications would not fail due to security violations during their normal use.
  - *Minimize administrator effort* by automating the development of file labels and integrity policies.
  - *Eliminate user prompts.* Security mechanisms that require frequent security decisions from users don't work well in practice for two reasons. First, users get tired and don't cooperate. Second, these user

interactions become the basis for social engineering attacks by malware writers.

- *Reduce reliance on trusted applications* so as to provide better overall assurance.

1.2. Salient Features

The principal elements of our approach that enables us to achieve the above goals are summarized below:

- *Flexible decomposition of high-level policies into low-level policies.* Traditional approaches for information-flow based integrity, such as the Biba integrity model [6], associate a label with an object (or subject) at the time of their creation, and this label does not change during the lifetime of the object or subject. In other words, these labels effectively define the access policies: a subject is permitted to read (write) an object only if the subject's integrity is equal to or lower (equal to or higher) than that of the object. In contrast, we distinguish between *labels,* which are a judgment of the trustworthiness of an object (or subject), from *policies* that state whether a certain read or write access should be permitted. Based on this separation, our approach (described in Section 2) allows integrity levels of objects or subjects to change over their lifetime. Moreover, "read-down" and "write-up" conflicts are resolved differently for different objects and subjects. These factors provide flexibility in developing low-level policies that preserve system integrity without unduly impacting usability.
- *Automated analysis for generating enforceable policies.* Given the large number of objects (hundreds of thousands) and subjects (thousands), manual setting of policies for every object/subject pair is impractical. In Section 3, we therefore develop techniques that utilize an analysis of access patterns observed on an unprotected system to automatically derive policies. This analysis can also be used to automatically complete the set of integrity-critical applications, starting from a partial list provided by a policy developer.

  As we show in Section 3.3, *our technique is sound, i.e., it will generate policies that preserve system integrity, even if the access logs used in analysis are compromised by malicious applications running on the unprotected system.* However, corrupted logs can compromise system availability.
- *A flexible enforcement framework.* Our enforcement framework, described in Section 5, consists of a small, security-critical enforcement component that resides in the OS kernel, and a user-level component that incorporates more complex features that enhance functionality without impacting security. This framework also incorporates features needed for learning and synthesizing policies for new applications.
- *Mechanisms for limiting trust.* There are some instances when high-integrity applications should be allowed to access low-integrity files. In Section 4, We develop techniques that enable such exceptions to be restricted. Our techniques typically have the effect of distinguishing between code/configuration inputs from data inputs, and ensuring that exceptions are made only for data inputs. Using these mechanisms, we describe how we can limit the amount of trust placed on important applications such as software installers, web browsers and email handlers, and file utilities.

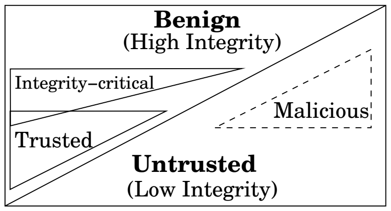

Figure 1. Classification of Applications

We have implemented our technique on desktop systems running RedHat/Ubuntu Linux, consisting of several hundred benign software packages and a few tens untrusted packages, the evaluation shows that the approach is practical, and does not impose significant usability problems. It is also effective in preventing installation of most malware packages and detection (and blocking) of malicious actions performed by stealthy malware.

2. Policy Framework

2.1. Trust and Integrity Levels

Figure 1 illustrates the integrity and trust levels used in our framework. To simplify the presentation, we use just two integrity levels: *high* and *low.* Integrity labels are associated with all objects in the system, including files, devices, communication channels, etc. A subset of high-integrity objects need to be identified as integrity-critical, which provide the basis for defining system integrity:

Definition 1 (System Integrity) *We say that system integrity is preserved as long as all integrity-critical objects have high integrity labels.*

The set of integrity-critical objects is externally specified by a system administrator, or better, by an OS distribution developer. It is important to point out that the integrity-critical list need not be comprehensive: if objects are left out of this list, our technique will automatically detect them using the analysis described in Section 3, as long as these objects are accessed after the kernel module enforcing *PPI* policies has begun execution. In particular, objects that are accessed during early phases of the boot process, as well those objects that are accessed by *PPI*, must be explicitly included in the integrity-critical set. On Linux, this (minimal) set of integrity-critical objects includes all the files within `/boot`, the binaries used by `init` and the files used by *PPI*, as well as devices such as the hard disk.

When referring to applications, we use the term "trust level" instead of "integrity level." Benign applications are those that are known to be benign, and may include applications that are obtained from a trusted source such as an OS vendor. Hence the files that constitute benign applications will have high integrity. Moreover, benign applications will remain trustworthy (i.e., produce high-integrity outputs) as long as they are never exposed to low-integrity (and hence potentially malicious) inputs. A subset of benign applications may be designated as *trusted.* These applications are deemed to sufficiently validate their inputs that they can produce high-integrity outputs even when some of their inputs have low-integrity. Untrusted applications are those that are obtained from untrusted sources, e.g., downloaded from an unknown website or those arriving via email from unauthenticated sources. An unspecified subset of these applications may be malicious.

Since trusted applications are being exempted from information flow policies, it is important that only a small number of well-tested applications are designated this way. In addition, the scope and purpose of this trust should be minimized as much as possible. We defer these issues until Section 4.

2.2. Integrity Labels Versus Policies

Given our goal of preserving system integrity, the Biba model is an obvious starting point [6]. However, a traditional interpretation of multi-level security (MLS) can lead to a rigid system that is difficult to use. To address this problem, we distinguish between *integrity labels* and *policies.* In our view, an integrity label on a file simply indicates whether its content is trustworthy, but does not dictate whether trustworthiness must be preserved, say, by preventing low-integrity subjects from writing to that file. A policy, on the other hand, indicates whether a certain read or write access should be permitted. This separation yields flexibility in developing policies that preserve system integrity without unduly impacting usability. For instance, we have the following choices for policies when a high-integrity subject (process) attempts to read a low-integrity object:

- **deny:** deny the access
- **downgrade:** downgrade the process to low-integrity
- **trust:** allow the access without downgrading the process, trusting the process to protect itself

The following examples illustrate the need for this flexibility. Consider a utility such as `cp` that accesses high-integrity objects in some runs (e.g., copy `/etc/passwd`), and accesses low-integrity objects in other runs (e.g., copy user files). Downgrading (the second alternative above), which corresponds to the low-water mark (LOMAC) policy [6, 7, 12], permits such dual use. However, this choice of downgrading is inappropriate in some cases, and leads to the well-known *self-revocation* problem: consider a process that has opened a high-integrity file H for writing, and subsequently attempts to read a low-integrity file L. If the process is downgraded at this point, we need to revoke its permissions to H. Applications typically assume that access permissions cannot be revoked, and hence may not handle the resulting access errors gracefully. On the other hand, if we deny the read access to L, it is likely to be better handled by the application.

To justify the third choice, consider an `SSH` server that reads low-integrity data from remote clients. The server code anticipates that clients may be malicious, and can reasonably be expected to protect itself adequately, thereby ensuring that the low-integrity input will not corrupt any high-integrity outputs. In contrast, the other two choices (deny or downgrade) will prevent the server from carrying out its function.

Analogous to the choices above, the following options are available when a low-integrity process attempts to write a high-integrity file (i.e., a file containing trustworthy data).

- **deny:** deny the access
- **downgrade:** downgrade the object to low-integrity, and allow the write operation
- **redirect:** redirect the access to a file f so that it accesses another file f_u instead. All subsequent accesses by an untrusted application to f will be redirected to f_u, while accesses by a benign application won't be redirected.

To justify the second choice, consider a file that is created by copying a high-integrity file. By default, the copy would have a high-integrity label, but if the copy is subsequently used only in low-integrity applications, downgrading it is the best option, as it would permit this use. As a second example, consider a commonly executed shell command such as `cat x > y`. Here, the shell will first create the output file `y` before `cat` is launched. If the shell has high integrity, then `y` would be created with high integrity, but subsequently, if `x` turns out to have low integrity, the best option is to downgrade `y` rather than to deny access to `x`. On the other hand, if a file is known to be used by high-integrity applications, it should not be downgraded, and the write access must be denied.

To justify the third choice, consider a benign application that is sometimes used with untrusted objects, e.g., a word-processor that is used on a file from the Internet. During its run, this application may need to modify its preferences file, which would have a high-integrity label, since the application itself is benign. Denying this write operation can cause the word-processor to fail. Permitting the access would lower the integrity of the preferences file, leading to all future runs of the word-processor to have low integrity. However, by redirecting accesses to a low-integrity copy of the preferences file, both problems can be avoided.

In summary, there are several ways to resolve potential integrity violations (conflicts), and different resolutions are appropriate for different combinations of objects, subjects, and operations. Our (low-level) policy specifies, for each combination of object O and subject S, which of the six different choices mentioned above should be applied. The sheer number of objects and subjects on modern operating systems (e.g., $>$100K files in system directories on typical Linux distributions) makes manual policy development a daunting task. We have therefore developed techniques to automate this task in the next section.

Although we discussed only file read/write operations above, the same concepts are applicable to other operations, e.g., file renames, directory operations, device accesses, and inter-process communication. We omit the details here, covering them briefly in the implementation sections.

3. Automating Policy Development

The large number of objects and subjects in a modern OS distribution motivates automated policy development. We envision policy development to be undertaken by a security expert — for instance, a member of a Linux distribution development team. The goal of our analysis is to minimize the effort needed on the part of this expert. The input to the policy generation algorithm consists of:

- software package information, including package contents and dependencies,
- a list of untrusted packages and/or files
- the list of integrity-critical objects
- a log file that records resource accesses observed during normal operation on an unprotected system.

We use these to compute dependencies between objects and subjects in the system, based on which trust labels and policies are generated. *Ideally, all accesses observed in the log would be permitted by these policies* without generating user prompts or application failures, but in practice, some failures may be unavoidable. Naturally, it is more important to minimize failures of benign applications as opposed to untrusted ones. Given this goal of policy analysis, it becomes important to ensure coverage of all typical uses of most applications in the log, with particular emphasis on benign applications. Since usage is a function of users and their environments, better coverage can be obtained by analyzing multiple logs generated on different systems.

3.1. Computing Dependencies, Contexts and Labels

Critical to the success of our approach was the observation that software package information, such as those contained in RPM or Debian packages, can be leveraged for extracting subject/object dependencies. The package information indicates the contents (i.e., files) in each package, and the dependencies between different packages.

Since some of the dependences, such as those on configuration or data files, may not be present in the package database, we analyze the access log to extract additional dependency information. The dependencies can vary for the same application depending on the context in which it is used. For instance, when used by system processes (such as a boot-time script), `bash` typically needs to be able to write high-integrity files, and hence must itself operate at a high-level of integrity. However, when used by normal users, it can downgrade itself when necessary to operate on untrusted files. In the former context, `bash` reads and writes only high-integrity files, whereas in the latter context, it may read or write a mixture of high and low integrity files. To identify such differences, we treat each program P as if it were several distinct programs, one corresponding to each of the following *execution contexts*: P_s that is used by system processes, P_a that is used by processes run by an administrator, and P_{u_i} for processes run by a normal user u_i. (The distinction between P_s and P_a is that in the second case, the program is run by a descendant of an administrator's login shell.) For simplicity, we will assume that there is only one normal user u.

Once the logs are generated, it is straight-forward to compute the set of files read, written, or executed by a program in each of the above contexts. In addition, for each program and file read (written or executed) by that program, we compute the fraction of runs of that program in each context during which the file was read (written, or executed).

Deriving Initial Object Labels. Initial object labels are derived using the following steps. The assumption behind this algorithm is that all packages and files on the system are benign unless specified otherwise:

- Mark all packages that depend on untrusted packages as untrusted.
- Label all objects that belong to untrusted packages (or are explicitly identified as untrusted) with low integrity.
- Label any object that was written by an untrusted subject (i.e., a process whose executable or one of its libraries are from low-integrity files) with low integrity.
- Label all other objects as having high integrity.

We reiterate that object labels do not dictate policy: an object may have a high label initially, but the policy synthesized using the algorithm below may permit it to be written by an untrusted subject, which would cause its label to become low.

3.2. Policy Generation

The policy generation algorithm consists of four phases as described below. The first phase identifies the set of "preserve-high" objects, i.e., objects whose integrity needs to be preserved. In the second phase, we generate the low-level policies for each (subject, object, access) combination, reflecting one of the policy choices described in Section 2.2. Phase III refines the policies by simulating the accesses contained in the log. Phase IV consists of ongoing policy refinement, as new applications are installed.

Phase I: Identification of objects whose integrity needs to be preserved.

1. *Initialize:*
   - For every package P such that an integrity-critical package depends on it, add P to the set of integrity-critical packages.
   - For every object that belongs to an integrity-critical package (or object that is explicitly labeled as integrity-critical), mark it as "preserve-high."
   - For every program P that is ever executed in the system context, mark P as preserve-high.
2. *Propagate from object to subject.* If a program P writes an object Q that is marked preserve-high, then mark P as preserve-high.
3. *Propagate from subject to object.* If a program P reads or executes an object Q then mark Q as preserve-high if any of the following hold:
   (a) P_s (i.e., P operating in system context) reads/executes Q.
   (b) P_a reads or executes object Q in non-negligible fraction of runs, say, over 10% of the runs.
   (c) P is marked preserve-high and P_u almost always reads or executes Q, say, in over 95% of the runs.

   Every program that is run in system context is expected to be run in high-integrity mode, and hence the first rule. Most activities performed in administrator context have the potential to impact system integrity, and hence most of these activities should be performed in high integrity mode, and hence the second rule. For the third rule, if a benign program P almost always reads or executes a specific file Q, then, if Q has low integrity, it will prevent any use of P in high integrity mode. It is unlikely that a benign program would be installed on a system in such a way that it is only executed at low-integrity level, and hence the third rule.
4. *Repeat the previous two steps until a fixpoint is reached.*

If any low-integrity file gets marked as preserve-high, there is a conflict and the policy generation cannot proceed until it is manually resolved. Such a conflict is indicative of an error in the input to the policy generation algorithm, e.g., a software package commonly used in system administration has been labeled as untrusted.

Phase II: Resolution of conflicting accesses. This phase identifies which of the policy choices discussed in Section 2.2 should be applied to each conflicting access involving (subject, object, access).

- **Deny policy:** For every object labeled preserve-high, the default policy is to permit reads by any subject but deny writes by low-integrity subjects. Similarly, for every object labeled with low-integrity, the default policy is to permit reads by low-integrity subjects but deny reads by high-integrity subjects. Exceptions to these defaults are made as described below, depending upon the program executed by a subject, and its trust level.
- **Downgrade subject policy:** A high-integrity subject P running in context c will be permitted to downgrade itself to low-integrity if there are runs in the log file where P_c read a low integrity file, and did not write any high integrity objects subsequently. Such runs show that P_c can run successfully, without experiencing security violations. If there are no such runs, then the downgrade policy is not used for P_c.

  Note that at runtime, a subject running P_c may still be denied read access if it has already opened an object O such that the policy associated with O prevents its label from being downgraded.

Finally, note that the use of context makes the downgrade policy more flexible. For instance, we may permit `bash` to downgrade itself when running in user mode, but not when it is run in system mode.

- **Trust policy:** Each subject P that reads a low-integrity object and writes to an object marked preserve-high is a candidate for the "trust" policy. Such candidates are listed, and the policy developer needs to accept this choice. If this choice is not accepted, the log analyzer lists those operations from the log that would be disallowed as a result of this decision.
- **Downgrade object policy:** Any object that is not marked as preserve-high can be downgraded when it is overwritten by a low-integrity subject.
- **Redirect policy:** A redirect policy is applied to the combination $(P, O, write)$ if (a) O is marked preserve-high, (b) P reads O in almost every run, and (c) P writes O in a non-negligible fraction of runs [1].

Phase III: Log simulation and policy refinement. The algorithm described above did not take into account that file labels will change as the operations contained in the log file are performed. (If we did not make the simplifying assumption that the labels are static, then the analysis would become too complex due to mutual dependencies between policy choices and file labels.) To rectify this problem, we "simulate" the accesses found in the log. We track the integrity levels of objects and subjects during this simulation, and report all accesses that cause a violation of the policy generated in the previous step. The violation reports are aggregated based on the subject (or object), and are sorted in decreasing order of the number of occurrences, i.e., the report lists the subject (or object) with the highest number violations first. Subjects with high conflict counts are suggestive of programs that may need to be trusted, or untrusted programs that cannot be used.

Based on the conflict report, the policy developer may refine the set of trusted, benign, or untrusted programs. If so, the analysis is redone. In general, more than one iteration of refinement may be needed, although in our experience, one iteration has proven to be sufficient.

Phase IV: Policy generation for new applications. New files get created in the system. In addition, new applications may become available over time. In both cases, we cannot rely on any "logs" to generate policies for them. Our approach is as follows.

For objects that are created after policy deployment, their labels will be set to be the same as that of the subject that created them. The default policy for such newly created objects is that their labels can be downgraded when they are written by lower integrity subjects. In addition, accesses to these objects are logged. The resulting log is analyzed periodically using the same criteria described above to refine the initial policy. For instance, if the object is used repeatedly by high-integrity subjects, the policy would be set so that writes by lower-integrity subjects are denied.

If a new software package is installed, labels for the objects in the package are computed from the trust level of the package, which must be specified at the time of installation. The policies for these files are then refined over time, as the package is used by the user.

3.3. Soundness of Policies

Recall that the policies derived above are based on accesses observed on an unprotected system. Being unprotected, it is possible for the log to have been compromised due to malicious untrusted code. Thus, an important question is whether the soundness of the derived policies is compromised due to the use of such logs. An important feature of our policy generation technique is that this does not happen. Thus, if the generated policies are enforced on a newly installed system, these policies will preserve its integrity.

Observation 2 *As long as the programs identified as* trusted *are indeed trustworthy, the policies generated above will preserve system integrity even if the access logs were compromised due to attacks.*

Proof sketch: Recall that preserving system integrity means that integrity-critical objects should never be written by low-integrity subjects. Observe that all integrity-critical objects are initialized to preserve-high in the first phase of the policy generation algorithm. The propagation steps in this phase can add to the set of objects marked preserve-high, but not remove any objects. In the next phase, note that "downgrade object" policy is applied only to those objects that aren't marked preserve-high. All other policy choices ensure that object labels will not be downgraded. Thus, the generated policy will ensure that the labels of integrity-critical objects remain high.

Observe that if the logs were compromised, far too many conflicts may be reported during policy generation. Worse, because the compromised logs may not reflect the behavior of programs on an uncompromised system, the generated policies may cause many accesses (not recorded in the log) to be denied, which can make the system unusable. Both these symptoms are sug-

[1] These three conditions characterize the access by most applications to their preference files — the context in which the redirect policy was motivated.

gestive of a compromised log file. The policy developer needs to obtain a new, uncompromised log and rerun the policy generation algorithm[2].

The above observation indicates that the primary weakness of *PPI* arises due to trusted programs. If they are incorrectly identified, or if they contain exploitable vulnerabilities, they can compromise end-user security objectives. This factor motivates features and techniques in the next section that limit and reduce the scope of trust.

4. Limiting Trust

Unlimited and unrestricted trust is often the weakest link in security, so we have incorporated features in *PPI* to reduce the scope and nature of trust placed on different programs. We describe these features below, followed by a discussion of how these features are used to address important applications such as software installers, browsers and email handlers, window systems, and so on.

Invulnerable and Flow-Aware Applications. All outputs of an *invulnerable applications* continue to have high integrity even after reading low-integrity inputs. An example would be an ssh server that can be trusted to protect itself from potentially malicious network inputs, and maintain its high integrity.

Flow-aware applications can simultaneously handle inputs with different integrity levels. They keep track of which inputs affect which outputs, and label the outputs appropriately. (Our enforcement framework provides the primitives for flow-aware applications to control the labels on their outputs.) Flow-awareness is natural for some applications such as web-browsers that already keep track of the associations between their input actions and output actions. (Web browsers use this type of information to enforce the "same origin policy [15].") Alternatively, automated techniques such as runtime taint-tracking [25, 36, 27] may be used to achieve flow-awareness.

A generic technique to mitigate the risk due to excessive trust is to deploy defenses against the most common ways of exploiting applications using malicious inputs, e.g., address-space randomization [5, 35] or taint-tracking[3] [25, 36, 27]. This technique can be combined with a more specific risk mitigation mechanism described below that limits trust to certain contexts.

[2]Recall that end-users are not expected to generate policies, so they won't experience the security failures that result due to compromised logs; and hence we don't expect this possibility to negatively impact end-user experience.

[3]Taint-tracking is preferable due to the weaknesses of ASR against local attacks.

Context-aware Trust. A key observation is that programs are rarely designed to accept untrusted inputs on every input channel. For instance, while an ssh server may be robust against malicious data received over the network, it cannot protect itself from malicious configuration files, shared libraries or executables. Our approach, therefore, is to limit trust to the specific input contexts in which an application is believed to be capable of protecting itself. For an ssh server, this may be captured by explicitly stating that it is invulnerable to inputs received on port 22.

With respect to files, one approach for limiting trust is to enumerate all the files read by an application in advance, and identify those that can have low integrity. This is far too cumbersome (or may not even be feasible) since the number of files read by an application may be very large (or unbounded). An alternative approach is to categorize a file as "data input" or "control input" (configuration or a library), and to permit a trusted application to read low-integrity data inputs but not control inputs. But manual identification of data and control inputs would be cumbersome. Instead, we rely on some of the properties of our policy synthesis techniques to achieve roughly the same effect. Specifically, note that configuration and library inputs will be read during practically every run of an application. As such, these files will be marked "preserve-high" in phase I of the policy generation algorithm, and hence the application will not be exposed to low-integrity configuration or library files[4].

4.1. Limiting Trust on Key Applications

Software Installers pose a particular challenge in the context of integrity protection. Previous techniques simply designated software installers as "trusted." This designation is problematic in the context of contemporary package management systems, where packages may contain arbitrary installation scripts that need to be run by the installer. During this run, they may need to modify files in system directories, and hence scripts cannot be run with low privileges.

We have developed a new approach to address this software installation problem. In our approach, the installer consists of two processes: a "worker" that runs as a low-integrity subject (but may have root privileges), and performs installation actions. To ensure

[4]This does not protect against the possibility that the application may, in a subsequent run, read a different configuration file. However, this is usually the result of running the application with a command-line option or an environment variable setting that causes it to read a different configuration/library file. These "inputs" are provided by a parent process, and hence are trusted, since the parent itself must have high-integrity in order for a child process to have high integrity.

that this low-integrity subject can overwrite system files if needed, a redirection policy is applied to all files written by this subject. A second high integrity "supervisor" subject runs after the first one completes. It verifies that the actions performed during installation were legitimate. In particular, it ensures that (a) the modifications made to the package management database are exactly those that were made to the file system, and (b) all the files installed are those that can be overwritten by a low-integrity subject. If the verification succeeds, the supervisor renames the redirected copies of files so that they replace their original versions. Otherwise, all the redirected copies are removed and the installation aborted.

Web Browser and Email Handler. Web browser and email client act as conduits for data received from the network. In our system, both web browser and email handler are considered flow-aware applications. Specifically, data received by a browser can be deemed high or low integrity based on the source of data and other factors such as the use of SSL. For the `Mozilla` browser used in our experiments, we built a small external program that uses the contents of a file "`downloads.rdf`" to correlate network inputs with the files written by the browser, and to label these files accordingly. We wrote a similar program for `pine` email reader.

X-Server and Other Desktop Daemons. GUI-based applications, called X-clients, need to access the X-Server. To ensure continued operation of benign as well as untrusted X-client applications, the X-Server should be made invulnerable on input channels where they accept data from untrusted clients. We mitigate the risk due to this trust in two ways. First, X-server is made invulnerable only on inputs received via sockets used to connect to an untrusted client. Second, we make use of the X security extension [33] to restrict low-integrity applications so that they cannot perpetrate attacks on other windows that would otherwise be possible.

Unfortunately, due to the design of the GNOME desktop system, there are some servers (e.g., gconfd) that are used by multiple X-clients and need to be trusted in order to obtain a working system. We are currently investigating techniques to limit trust on these applications. Some of the recent results from the SE-Linux project [31, 18] could be applicable in this context.

File Utilities. Applications that can run at multiple trust levels can sometimes introduce some usability issues, specifically, when they are used to operate on input files with varying trust levels. We modified `cp` and `mv` to make them flow-aware, so that the output files correctly inherit the label from the input files.

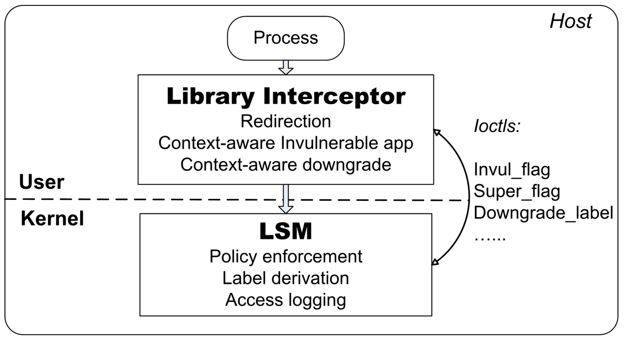

Figure 2. *PPI* **System Architecture**

5. Enforcement Framework

Our design is a hybrid system consisting of two components: a user-level library and a kernel-resident checker. Integrity-critical enforcement actions are performed by the kernel component, while "functionality enhancement" features are relegated to the library. For instance, the kernel component does not deal with redirection policy. Moreover, while it supports the notion of trusted subjects, it does not concern itself with mechanisms for limiting trust, which are provided by the user-level component. While the kernel enforcement component is always trusted, the user-level component is trusted only when it operates in the context of a high-integrity process.

In our implementation, the kernel level component is realized using LSM (Linux Security Modules) [34], which has now become part of the standard Linux kernel. We use the security hooks of LSM to enforce information flow policies. Although our policy framework allows for policies to be a function of objects as well as subjects, for simplicity, the policies enforced by the kernel component have limited dependence on subjects. (More flexible subject-dependent policies can be realized using the user-level component.) This enables kernel-enforced policies to be stored with objects using *extended file attributes* available on major Linux file systems (including ext2, xfs, and reiserfs). Policies as well as integrity labels are stored using these attributes. Specifically, a 3-bit attribute *integ_obj* is used to store the integrity level of a file. (For extensibility, our implementation uses eight integrity levels.) A 11-bit policy is associated with each file, consisting of two parts. The first part pertains to read and write operations performed on the file:

- *down_obj* (3 bits) indicates the lowest integrity level to which this object can be downgraded.

- *log_obj* (1 bit) indicates whether accesses to this object should be logged. This feature could be used for auditing. In our implementation, it provides the mechanism for generating the logs used for policy generation.

The second part of the policy pertains to the use of a file to instantiate a subject, i.e., when the file is executed. It consists of the following components:

- *down_sub* (3 bits) indicates the lowest integrity level to which a process that executes this object can be downgraded.
- *log_sub* (2 bits) indicates whether accesses made by this subject should be logged. A second bit indicates whether this policy should be inherited by descendants of a subject.
- *invul_sub* (1 bit) indicates if this subject is invulnerable. No distinction is made among various subclasses of trusted applications described in Section 4 — it is up to the user-level component to implement distinctions such as flow-awareness and context-awareness.
- *super_sub* (1 bit) allows a subject to modify the labels associated with objects in the system. Naturally, this capability should be highly restricted. In our implementation, there is one administrative program that has this capability.

When *PPI* system is initialized, the extended attributes associated with all the files are populated using the labels and policies generated using the techniques in Section 3. New files inherit the integrity of the subject creating them. The log bits are set by default, super_sub and invul_sub bits are cleared, and the down_sub and down_obj bits are set to zero. (A lower integrity level or a higher downgrade level may be requested by any subject.)

After a fork, the child inherits the parent's attributes, including its integrity level. After an exec, the integrity of the subject is reduced (but can never be increased) to that of the file being executed. The super_sub policy is inherited from an executable file only if the subject is at the highest possible integrity level. Finally, the invul_sub as well as log_sub attributes are set from the executable file.

Handling Devices, IPC, and File Renaming. Devices typically need special treatment since they are not persistent in the same sense as files. The integrity and down_obj labels of devices such as `/dev/kmem`, `/dev/mem`, and `/dev/sda*` are set to be 111 to ensure that only the highest integrity subjects can modify them. Devices such as `/dev/null` and `/dev/zero` are treated as having low-integrity for writes, and high integrity for reads. The integrity labels of devices such as `/dev/tty*` and `/dev/pty/*` are derived from that of the associated login processes, in a manner similar to that of SELinux.

IPC and socket communication are treated as communication channels between subjects. As such, they inherit labels from the subjects that they connect. Moreover, since most of these communication mechanisms are bidirectional, subjects interconnected using them have identical security attributes[5]. The kernel enforcement component keeps track of "groups" of subjects interconnected using IPC and socket communication so that operations such as downgrading can be performed safely.

Finally, file renaming operations are deemed as a write operation on the file object.

Implementing Subject Downgrading. Subjects are downgraded at runtime only if doing so will not result in revocation of privileges. Specifically, the user-level component indicates, at the time of opening a file for read, whether a higher integrity subject is willing to downgrade to a lower level. The kernel component permits the downgrade if (a) the file can be successfully opened, (b) the subject does not have higher integrity files that it is writing into, and (c) all other subjects with which this subject has IPC can also be downgraded.

User-Level Component. The user-level component is implemented by intercepting system calls in `glibc`. Since the policies themselves are application-specific, their implementation is kept in a separate library. The user-level communicates with the kernel level using ioctl's to implement complex policies. We already described how "downgrade subject" policy is implemented through such coordination. It also supports *context-aware trust policies:* the user level determines whether a trusted application is invulnerable (or flow-aware) in a certain context, and if so requests an open without downgrading its integrity. For flow-aware applications, the user-layer communicates to the kernel layer if files should be opened with a lower integrity level than that of the subject. The user level is also responsible for implementing the redirection policy. The kernel layer is unaware of this policy, and simply treats it as a combination of a read operation on the original file, and a write operation on a new file.

[5]An exception occurs in the case of trusted subjects that are invulnerable to inputs on a communication channel: in this case, the trusted process can continue to maintain its integrity level and other security attributes regardless of the security attributes of the subject on the other end of the communication channel.

6. Evaluation

We initially implemented *PPI* on CentOS and subsequently migrated to a Ubuntu system. Due to time limitations, some of our evaluations were performed on CentOS while others were performed on Ubuntu. The specifics of these two systems are as follows:

- CentOS 4.4 distribution of Linux with kernel version 2.6.9. The test machine has 693 rpm packages and 205k files in total.
- Ubuntu 7.10 distribution of Linux with kernel version 2.6.22-14. The test machine has 1164 dpkg packages and 159k files in total.

6.1. Experience

Policy Generation on Ubuntu. For generating policies, we used an access log collected over a period of 4 days on a personal laptop computer. We also carried out administrative tasks such as installing software, running backups, etc. The log file we obtained was around 1GB.

The set of initial integrity-critical file objects include files within `/boot`, `/etc/init.d/`, `/dev/sda` and `/dev/kmem`. We identified 26 untrusted application packages, which include:

- *Media players:* mplayer and mpglen
- *Games:* gnome-games, crafty and angband
- *Instant messengers:* pidgin
- *Emulators:* apple2 and dosemu
- *File utilities:* rar, unrar and glimpse
- *X utilities:* axe
- *Java applications:* jta, sun-java6-bin and sun-java-jre

Based on the above initial configurations to log analysis, we performed the procedures described in Section 3 for label computation and policy generation.

Among all the files (159K) and dpkg packages (1164) on the system, the initial labels of 783 files (including files that belong to 26 untrusted packages and those written by them) were set to low integrity, while all the others are labeled high integrity initially.

Then we moved on to the policy generation phase. The number of subjects and objects that were assigned to different policy choices across different phases are summarized in Figure 3. In Phase I, the log analysis determined 73305 files (934 packages) that need to be marked "preserve-high." In Phase II, the analysis identified which of the six policy choices should be applied to each conflicting accesses. As a result, 168 programs that are exclusively run in system context were assigned "subject-deny" policy, and they won't be allowed to read untrusted input. 4 programs (Xorg, dbus-daemon, gconftool-2, gnome-session) were marked "trusted," and they would retain high integrity level even when exposed to low-integrity input in certain channels, for instance, `/tmp/.X11-unix/X0` in the case of Xorg. The rest programs (6905) can be downgraded when reading low-integrity input. Correspondingly, for file objects, deny policies were applied to 73305 integrity critical objects, while the others can be downgraded[6]. Finally, the analysis identified 15 file objects for redirection policy, including files such as the preferences files for gedit editor.

Phase III used the initial policy configuration from Phase II. It reported 66 violations due to object and subject labels being downgraded in a running system. (A few thousand object and subject downgrades were observed.) Of all the violations, `trackerd` accounted for 51 conflicts. Another ten conflicts were due to `nautilus`, `gconfd-2`, and `bonono-activation-server`. After an analysis of the conflict report, it was determined that these four applications needed to be classified as trusted. Finally, four conflicts arose because `bash` could not write to the history file. This was resolved by using the redirect policy.

In addition, as described earlier, Mozilla and pine were also identified to be trusted applications. (One could of course avoid this for a browser by running two instances, one operating in high-integrity and used to access high-integrity web sites, and the other in low-integrity to access untrusted web sites.) Most of the trusted applications listed above should be made flow-aware, so they label their outputs appropriately, based on the integrity of their inputs.

PPI ***Experience during Runtime.*** After we plugged in the policies generated in the previous step, we ran *PPI* in normal enforcing mode for several days. The system booted up without problems, which indicated that none of the init related programs were affected by low integrity input. We first ran all the normal administration type of tasks using root privilege, such as checking disk space, configuring network interfaces, etc. Then we logged in as normal user, and worked on the system as a typical user would. We also used all of the untrusted applications installed on the system. None of these activities raised any security violations. (We were able to create violations on purpose, e.g., by trying to edit a high and low integrity file at the same time using a benign editor.)

One slight problem is the side-effect of redirection

[6]It should be noted that one of the main reasons for so many files being labeled downgradable is that the log was collected over a relatively short period of time, during which many of the applications were not exercised. Thus, we should view the policies on these remaining files as "initial" policies that would be further refined in Phase IV of policy development. (Implementation of Phase IV is ongoing work.)

	Phase I		Phase II		Phase III	
	subject	object	subject	object	subject	object
subject-deny			168		168	
subject-downgrade			6905		6905	
subject-trust			4		8	
object-deny		73305		73305		73305
object-downgrade		86185		86185		86185
object-redirect				15		16

Figure 3. ***PPI*** **Policy Generation in Different Phases.**

policies: a duplicate copy of many of the preference files will be created as a result. One option is to periodically delete the low-integrity version of these files.

6.2. Effectiveness Against Malicious Applications

Our integrity policy described in Section 3 provides effective defense against malware attacks.

- Linux rootkits. In this experiment, we downloaded around 10 up-to-date rootkits from [1]. Since our browser is flow-aware, it checked the source of the downloaded software, and marked them as untrusted. User level rootkits such as bobkit, tuxkit, and lrk5 required an explicit installation phase. *PPI* reported permission violations such as deletion of /etc/rc.sysinit and /bin/ps, and hence their installation failed.

  Kernel level rootkits in the form of kernel modules are prevalent nowadays and are more difficult to detect. We downloaded, compiled, and installed one such rootkit, adore-ng. Since the initial download was low-integrity, the kernel module was also labeled with low-integrity. Since *PPI* does not permit loading of kernel modules with low integrity, this rootkit failed. Since only high integrity subjects are allowed to write to /dev/kmem, another kernel rootkit moodnt failed with the error message ".D'ho! Impossible aprire kmem."
- Installation of *"Malicious" rpm package.* The Fedora package buildsystem [11] suggest three possible attack scenarios from the malicious package writer. Of these, a malicious rpm-scriptlet is a serious threat. To test the effectiveness of *PPI* under this threat, we crafted a "malicious" rpm package. This package is named `glibsys.rpm`. During the installation phase, the package tried to overwrite system files `/lib/libc.so` and `/bin/gcc`. These violations are captured by our system, and the installation aborted cleanly.
- Race condition attack. We crafted a piece of malware which employed a typical race condition attack. The attack we created is the classic TOCTTOU race condition [8], allowing a malicious process racing against a benign process in writing a high-integrity file (/etc/passwd) using a race condition by creating a symbolic link. This attack was successful on an unprotected system, but it was defeated by *PPI*, since the `follow_link` operation on the low-integrity symlink downgraded the benign process and the write was disallowed.
- Indirect attack. In this attack, we created another piece of malware that first created an executable with an enticing name and waited for users on the system to run it. The attack did not work as *PPI* automatically downgraded the process running the low integrity executable, and as a result, it could not overwrite any of the files in the system that can damage system integrity.
- Malformed data input. Similar to the above example, a malformed jpeg file was downloaded from some unknown source, so *PPI* marked it as low-integrity. When an image viewer opened it, although it was compromised, it was running in low-integrity mode, and hence its subsequent malicious actions failed.

6.3. Usability of PPI Versus Low Watermark

In order to better understand how *PPI* provides improved usability, we implemented a prototype version of Low watermark model using LSM, and applied the prototype to exactly the same host environment with the same initial labeling and policy configurations (including invulnerable applications). We used the test environment for a period of one day and observed the violations in the following several types:

- Because of lack of object downgrade policy, in Low watermark model, gzip and ggv had difficulty in completing their jobs when handling low integrity files. In the case of gzip, it first ran with high integrity, created an output file, then downgraded on reading low-integrity input. Subsequently, gzip tried to change permissions on the output file, which was denied due to the fact that the file was at high integrity while

	simple syscall	read	write	stat	open/close	select(100)	pipe latency
Orignal	1.6936	2.1882	1.8670	7.9352	11.4859	27.2026	37.4196
PPI	1.6862	2.3541	1.9899	15.0348	15.0348	28.5309	38.5261

Figure 4. Microbechmark Result using LMbench. All numbers are in microseconds.

the subject had been downgraded.

- With *PPI*, editors such as vi, gedit and gimp could be used to edit low-integrity as well as high-integrity files. With Low-watermark policy, the applications experienced a runtime error, if the first run of the application was performed with high-integrity input. In this case, the preference files were marked high-integrity. When the editor was subsequently used on a low-integrity file, it was downgraded, and its subsequent access to update the preference file was denied. If the the first run was performed with low-integrity input, then the preference file was created with low integrity, which meant that every future run of the editor will be downgraded. In contrast, the use of log-based analysis in *PPI* enables these editors to work properly in both scenarios.
- As mentioned earlier, shell redirections typically cause problems due to self-revocation with Low-watermark model. For instance, when executing a command such as `cat in > out`, the shell, typically running at high-integrity, creates the file `out` with high-integrity. If `in` is a low-integrity file, then `cat` will be downgraded on reading it, and then its attempt to write to `out` will be denied.

6.4. Performance Overheads

We present both the microbenchmark and macrobenchmark results for *PPI*. For microbenchmark evaluation, we used LMbench version 3 [23] to check the performance degradation of popular system calls. The results are summarized in Figure 4. We observed that *PPI* did not introduce noticeable overhead for most system calls except for open (and other similar system calls such as stat). For macrobenchmark, we measured 3 typical applications running within *PPI* during runtime. As illustrated in the Figure 5, the runtime overhead for applications in *PPI* is about 5% or less.

6.5. Limitations

Our approach cannot support arbitrary untrusted software. Some software, by its very nature, may need resource accesses that cannot safely be granted to untrusted applications. Our results show that for the type of programs that tend to be downloaded from untrusted sources, our approach is indeed applicable.

Our work does not focus on confidentiality or availability, but it still contributes to them in two ways. First, solutions for confidentiality and availability must build on solutions for integrity. Second, our techniques halt malware that exploits integrity violations to attack confidentiality; for example, by preventing a rootkit from installing itself, we also prevent it from subsequently harvesting and sending confidential account information. But no protection is provided from malware that targets violation of confidentiality without violating integrity.

	Original	*PPI* Mode	
	Time	Time	Overhead
gzip	1.269	1.271	0.1%
xpdf	2.476	2.604	5.1%
make	22.467	23.345	3.8%

Figure 5. Application Performance Overhead. All numbers are in seconds averaged across 10 runs.

7. Related Work

Information Flow Based Systems. *Biba* model [7] has a strict "no read down and no write up" policy. The low-water mark model [7] relaxes this strict policy to permit subjects to be downgraded when they read low-integrity inputs. *LOMAC* [12], a prototype implementation of low-water mark model on Linux, addresses the "self-revocation" problem to a certain extent: a group of processes that share the same IPC can continue to communicate after one of the processes is downgraded by having the entire group downgraded, but the problem still remains for files. *SLIM (Simple Linux Integrity Model)* [28] is part of the IBM research trusted client project, and is also based on the LOMAC model. It also incorporates the Caernarvon model [17], which supports verified trusted programs and limits the trust on these programs by separating read and execute privileges. The features developed in this paper are more general in this regard, allowing distinctions between data input and control input, and so on.

IX [22] is an experimental MLS variant of Unix. It uses dynamic labels on processes and files to track information flow for providing privacy and integrity. In contrast, our technique generalizes the LOMAC model by offering several other policy choices, which we have shown to be helpful for improving usability. Other important distinctions between these works and ours are that we decouple policies from labels, and provide automated techniques for policy development.

Windows Vista enforces only the "no write up" part of an information flow policy, "no read down" is not enforced as it causes usability problems. Unfortunately, malware writers can adapt their malware to defeat this protection, as discussed in the introduction. In contrast, *Back to the Future* system [14] enforces only the "no read down" policy. Its main advantages are that it can recognize any attempt by malware to inject itself into inputs consumed by trusted applications, and the ability to rollback malware effects. A drawback is that any attempt to "use" the output of an untrusted (but not malicious) application would require user intervention. It can be difficult for users to judge whether such inputs are "safe" and respond correctly to such prompts. Secondly, malware can freely overwrite critical files, which need to be "recovered" when the data is subsequently accessed — a potentially time-consuming operation.

Safe Execution Environments [3, 30, 37] and virtual machines [32, 9, 4] rely on isolation to confine untrusted processes. While isolation is an effective protection technique, maintaining multiple isolated working environments is not very convenient for users. In particular, objects such as files that need to be accessed by untrusted code have to be copied into and/or out of the isolated environment each time.

Li et al [20] also address the problem of making mandatory access control usable by focusing on techniques for policy development. However, their focus is on servers exposed to network attacks, as opposed to untrusted software threats on workstations. The nature of the threat (remote attacks versus adaptive malware) is quite different, causing them to focus on techniques that are quite different from ours. For instance, they don't protect user files, while we consider corrupting of user files to be a very powerful attack vector in our context. Moreover, they do not consider the problem of securing software installations, or provide analysis techniques that can leverage resource access logs to generate policies. Nevertheless, there are some similarities as well: we have been able to use their notion of limiting trust to certain network channels. In addition, we provide a refinement of this notion in the context of files.

All of the above works were based on centralized policies, which are less flexible than decentralized information-flow control (DIFC) policies. DIFC policies allow applications to control how their data is used. In this regard, JFlow [24] is a language level approach. Asbestos [10] and Hi-Star [38] are new operating system projects that have been architected with information flow mechanisms incorporated in their design. Flume [19] is focused on implementing an extension to existing operating systems to provide process level DIFC. Like most other previous works in information-flow based security, these projects too focus on mechanisms, whereas our focus has been on generating the policies needed to build working systems.

SELinux, Sandboxing and Related Techniques. Several techniques have been developed for sandboxing [13, 2, 26]. Model-carrying code [29] is focused on the problem of policy development, and provides a framework for code producers and code consumers to collaborate for developing policies that satisfy their security goals. Nevertheless, development of sandboxing policies that can robustly defend against adaptive malware is a challenge due to the ease of indirect attacks as described in the Introduction.

SELinux [21] uses domain and type enforcement (DTE) policies to confine programs. Their main focus has been on servers, and they have developed very detailed policies aimed at providing the least privilege needed by such applications. Systrace project [26] has also developed system-call based sandboxing policies for several applications, and is widely used in FreeBSD. Neither approach ensures system integrity by design. SELinux as well as Systrace can log accesses made during trial runs of an application, and use it as the basis to generate a policy for that application. Their policy generation technique is useful for trusted code, such as servers, but would be dangerous for untrusted applications.

Whereas our focus is on generating policies that ensure integrity, other researchers have worked on the complementary problem of determining whether a given policy guarantees system integrity [16].

8. Conclusion

In this paper, we presented techniques for proactive integrity protection that scales to a modern operating system distribution. By enforcing information flow policies, our approach provides positive assurances against malware from damaging system integrity. One of the central problems in developing practical systems based on such mandatory access control policies has been the complexity of policy development. We have developed several techniques to automate the generation of low level information flow policies from data contained in software package managers, and logs that capture normal usage of these systems. Our experimental results show that the technique is efficient, can provide protection from most malware, and does not unduly impact system usability.

References

[1] Linux rootkits. http://www.eviltime.com.

[2] A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. In *USENIX Security Symposium*, 2000.

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

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. 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, Oct. 19–22 2003. ACM Press.

[5] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: an efficent approach to combat a broad range of memory error exploits. In *Proceedings of the 12th Usenix Security Symposium*, Washington, D.C., August 2003.

[6] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, June 1975.

[7] 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.

[8] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2), 1996.

[9] J. Dike. A User-Mode port of the linux kernel. In *Proceedings of the 4th Annual Showcase and Conference (LINUX-00)*, pages 63–72, Berkeley, CA, Oct. 10–14 2000. The USENIX Association.

[10] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazires, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *20th Symposium on Operating Systems Principles (SOSP 2005)*, 2005.

[11] *The fedora.us buildsystem.* http://enrico-scholz.de/fedora.us-build/html/.

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

[13] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *USENIX Security Symposium*, 1996.

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

[15] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 737–744, New York, NY, USA, 2006. ACM.

[16] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[17] P. Karger, V. Austel, and D. Toll. Using a mandatory secrecy and integrity policy on smart cards and mobile devices. In *EUROSMART Security Conference*, pages 134–148, Marseilles, France, 2000.

[18] P. Karger, V. Austel, and D. Toll. Using gconf as an example of how to create an userspace object manager. page SELinux Symposium, 2007.

[19] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 321–334, New York, NY, USA, 2007. ACM.

[20] N. Li, Z. Mao, and H. Chen. Usable mandatory integrity protection for operating systems. In *IEEE Symposium on Security and Privacy*, 2007. To appear.

[21] P. A. Loscocco and S. D. Smalley. Meeting critical security objectives with security-enhanced linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.

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

[23] L. McVoy and C. Staelin. Lmbench. http://www.bitmover.com/lmbench/.

[24] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[25] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[26] N. Provos. Improving host security with system call policies. In *Proceedings of the 11th USENIX Security Symposium*, pages 257–272, 2003.

[27] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting general security attacks. In *IEEE/ACM International Symposium on Microarchitecture*, December 2006.

[28] D. Safford and M. Zohar. A trusted linux client (tlc), 2005.

[29] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating System Principles*, Bolton Landing, New York, October 2003.

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

[31] E. F. Walsh. Integrating xfree86 with security-enhanced linux. In *X Developers Conference*, Cambridge, MA, 2004.

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

[33] D. P. Wiggins. Security extension specification, version 7.0. Technical report, X Consortium, Inc., 1996.

[34] C. Wright, C. Cowan, J. Morris, S. Smalley, G. KroahHartman, s modules, and G. support. Linux security modules: General security support for the linux kernel, 2002.

[35] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Symposium on Reliable and Distributed Systems (SRDS)*, Florence, Italy, October 2003.

[36] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, August 2006.

[37] Y. Yu, F. Guo, S. Nanda, L. chung Lam, and T. 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.

[38] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazires. Making information flow explicit in histar. In *Seventh Symposium on Operating Systems Design and Implementation (OSDI06)*, 2006.