A Portable User-Level Approach for System-wide Integrity Protection[†]

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

ABSTRACT

In this paper, we develop an approach for protecting system integrity from untrusted code that may harbor sophisticated malware. We develop a novel dual-sandboxing architecture to confine not only untrusted, but also benign processes. Our sandboxes place only a few restrictions, thereby permitting most applications to function normally. Our implementation is performed entirely at the user-level, requiring no changes to the kernel. This enabled us to port the system easily from Linux to BSD. Our experimental results show that our approach preserves the usability of applications, while offering strong protection and good performance. Moreover, policy development is almost entirely automated, sparing users and administrators this cumbersome and difficult task.

1. Introduction

The state-of-practice in malware defense relies on reactive measures, such as virus scanning and software patches. While this practice may have been adequate in the past, it cannot cope with today's sophisticated malware that employ complex evasion and subversion techniques to overcome deployed defenses. It is thus important to develop principled defenses that provide reliable protection regardless of malware sophistication.

A natural (and perhaps the best studied) proactive defense is to sandbox potentially malicious code. This approach can be applied to software from *untrusted* sources [11], which may be malicious to begin with; or to software from *trusted* sources [16, 6, 20] that is benign to start with, but turns malicious due to an exploit. However, there are several challenges with sandboxing untrusted code:

- *Difficulty of policy development.* Experience with SELinux [16] and other projects [3, 22] show that policy development requires a great deal of expertise and effort. Moreover, policies that provide even modest protection from untrusted code can break many legitimate applications.
- *Subversion attacks on benign software.* Even highly restrictive policies can be inadequate, as malware can co-opt benign applications to carry out prohibited operations: Malware may trick a user to run a benign application in insecure ways or exploit vulnerabilities in benign applications to perform arbitrary actions.

ACSAC '13, December 09 - 13 2013, New Orleans, LA, USA

ACM 978-1-4503-2015-3/13/12 ...$15.00.

- *Difficulty of secure policy enforcement.* Non-bypassable policies usually have to be enforced in the OS kernel. They are usually much harder to develop than user-level defenses. Moreover, kernel-based solutions cannot be easily ported across different OSes, or even different versions of the same OS.

An alternative to sandboxing is isolated execution of untrusted code. One-way isolation [15, 23] permits untrusted software to read anything, but its outputs are held in isolation. Two-way isolation limits both reads and writes, holding the inputs as well as outputs of untrusted applications in an isolated environment. The app model on Android, Apple iOS, and Windows 8 sandbox are generally based on this two-way isolation model.

Isolation approaches provide stronger protection from malware since they block all interactions between untrusted and benign software, thereby preventing subversion attacks. They also provide much better usability because they permit sufficient access for most applications to work. However, they too have several significant drawbacks, especially in the desktop environment:

- *Fragmentation of user data.* Unlike sandboxing, which continues to support the model of a single namespace for all user data, isolation causes a fragmentation: user data is partitioned into two or more containers, each representing a disjoint namespace.
- *Inability to compose applications.* The hallmark of today's desktop OSes is the ability to compose applications together. UNIX pipelines represented one of the early examples of application composition. Other common forms of composition can happen through files or scripts, e.g., printing a spread sheet into a PDF file and then emailing this PDF file. Unfortunately, strict isolation prevents one application from interacting with any data (or code) of other applications, thus precluding composition.
- *No protection when isolation is breached.* Strict isolation may be breached either due to a policy relaxation, or through manual copying of files across isolation contexts. Any malware present in such files can subsequently damage the system.

In contrast, our approach combines the strengths of sandboxing and isolation of untrusted code, while avoiding most of their weaknesses. Like sandboxing, all user data is held within one name space, thereby providing a unified view. Like isolation, our approach preserves the usability of applications, and does not require significant policy development effort. At the same time, it avoids the weakness of isolation-based approaches, allowing most typical interactions between applications, while ensuring that system security isn't compromised by these interactions. An open-source implementation of our system is available [26].

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

Term	Explanation
malicious	intentionally violate policy, evade enforcement
untrusted	possibly malicious
benign program	non-malicious but may contain vulnerabilities
benign process	process whose code and inputs are benign, hence non-malicious

Figure 1: Key terminology

1.1 Approach Overview and Salient Features

Sophisticated malware can evade defenses using multi-step attacks, with each step performing a seemingly innocuous action. For instance, malware may simply deposit a shortcut on the desktop with a name of a commonly used application instead of writing files in system directories directly. It can wait until the user double-clicks on this shortcut and do its work. Alternatively, malware may deposit files that contain exploits for popular applications, with the actual damage inflicted when a curious user opens them. The first example involves a benign process executing code derived from a malicious source, while the second example involves a vulnerable benign application being compromised by malicious data.

To thwart all malware attacks regardless of the number of steps involved, we use integrity labels to systematically track the influence of untrusted sources on all files. Files coming from the OS vendor (and any other source that is trusted to be non-malicious) are given the label *benign* (Figure 1), while the remaining files are given the label *untrusted.* Note that *benign programs* may contain exploitable vulnerabilities, but only *untrusted programs* can be malicious, i.e., may intentionally violate policy and/or attempt to evade enforcement. Exploitation of vulnerabilities can cause benign programs to turn malicious. However, an exploit represents an intentional subversion of security policies, and hence cannot occur without the involvement of malicious entities. Consequently, *benign processes*, which are processes that have never been influenced by untrusted content, cannot be malicious. New files and processes created by benign processes can hence be labeled benign. Processes that execute untrusted code (or read untrusted inputs), are labeled as untrusted, as are the files created or written by them.

The core of our approach is information-flow based integrity preservation, similar to the Biba integrity model [5]. Our main contribution is that of solving the key challenges in adopting such a model to contemporary operating systems:

- *Secure information-flow tracking and policy enforcement without OS kernel changes.* Absence of kernel changes not only simplifies the implementation but also makes it possible to experiment with the large base of existing OS and application software. Moreover, it leads to a smaller TCB, and makes the implementation portable across OSes.
- *Preserving user experience.* Enforcement of mandatory access control (MAC) policies such as MLS or Biba model can often break existing applications, since many previously permitted operations are now disallowed by the MAC policy. Our approach incorporates several refinements to the basic information flow policy that preserve functionality without degrading integrity. As a result, our approach can preserve the user experience on contemporary OSes, as shown by our experiments.
- *Automating policy development.* Policy refinements often come with a steep price: they require substantial development efforts, typically for every application. Thus, protecting an entire OS distribution can become prohibitively expensive if policies have to be developed manually. We therefore present techniques that automate policy development in almost all cases.

We expand on these points further below.

1.1.1 Secure enforcement and tracking without OS changes

Our approach encodes integrity labels into file ownership and permission. In particular, untrusted files are those that are owned by a set of newly created untrusted userids, or are writable by these users. Untrusted processes are all run with an untrusted userid. This encoding enables us to leverage existing OS mechanisms for tracking and propagating integrity labels. In particular, note that files as well as child processes inherit their ownership from that of the process that created them. As a result, any file or process created by an untrusted process will have the label of untrusted.

Benign processes and files are characterized by their ownership by a userid other than an untrusted userid. In addition, benign files will have write permissions that make them unwritable by untrusted userids. As a result, files created by benign processes will have benign labels, once again ensuring correct propagation of labels.

In addition to tracking integrity labels, our userid-based encoding also provides the foundation for sound policy enforcement without OS kernel changes. Specifically, existing OS mechanisms can correctly enforce policies on untrusted processes: by virtue of our integrity label encoding, benign files have permission settings that make them unwritable by untrusted userids. Although we need to develop additional enforcement mechanisms for benign processes, e.g., to prevent them from reading untrusted files, this is a considerably simpler task than policy enforcement on untrusted code. In particular, challenges in secure policy enforcement arise mainly due to evasion attacks. Since benign processes cannot be malicious, they won't attempt evasion. Indeed, a simple yet secure implementation can be developed within the address space of a benign process, e.g., by replacing libc, which makes all system calls on behalf of a process, with a version that enforces the desired policies.

1.1.2 Preserving user experience

Increased security is usually achieved through stronger security policies. These stronger policies will invariably deny some (otherwise allowed) operations, thus impacting functionality. While careful policy development may reduce the scope of functionality loss, experience with SELinux [16] and other projects [3, 22] show that (a) the effort and expertise involved in developing good policies is considerable, and (b) the resulting policies can still lead to unacceptable loss of functionality (or security). The fundamental problem is that finding the "boundary" between legitimate and insecure behaviors can be very hard. For instance, consider identifying the complete set of files that must be protected to ensure host integrity. An overly general list will cause untrusted applications to fail because their file accesses are denied, while omissions in this list will impact benign system operations. If untrusted software is prevented from writing any files within a user's home directory, this can affect its usability. If, on the other hand, it is permitted to write any file, it may be able to install backdoors into the user's account, e.g., by modifying files that are automatically executed with user's privileges, such as the `.bashrc` file.

We overcome the dilemma with a novel *dual-sandbox architecture*. The first of these sandboxes performs eager policy enforcement. To minimize breaking legitimate functionality, it blocks only those operations that can cause irreparable damage, e.g., overwriting an existing benign file. This sandbox, called untrusted sandbox (U), needs to be secure against any attempts to circumvent it.

Operations with unclear security impact, such as the creation of new files, are left alone by the second sandbox, called benign sandbox (B). While these actions could very well be malicious, there isn't enough information to make that conclusion with confidence. Hence, we rely on B to observe subsequent effects of this action to determine if it needed to be stopped. For instance, if such a file is

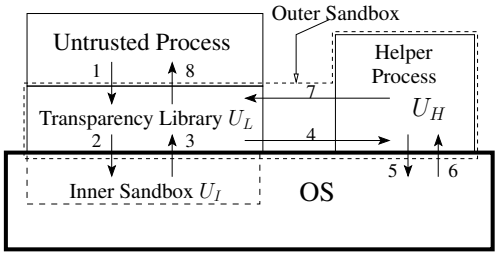

Figure 2: Untrusted sandbox

used by a benign process, it could compromise the benign process. B prevents such use.

Our dual-sandbox architecture achieves several important goals: First, it provides robust enforcement of complex policies without requiring OS kernel modifications. Second, it preserves functionality of both benign and untrusted applications by implementing many important transparency features, so that security benefits of our approach can be achieved without requiring changes to applications, or the way in which users use them.

1.1.3 Automating policy development

To build a practical system that preserves user experience, we need as much (if not more) emphasis on the policies as on the enforcement mechanisms. However, policy development is often a manual process that requires careful consideration of every application and file on the system. Given that a typical Linux system may have thousands of applications and many tens of thousands of files, this becomes a truly daunting task. We have therefore developed a procedure for classifying files into different categories: code, configuration, preference and data. Based on this inference, we provide a detailed policy that works without needing manual analysis of applications or file residing on the system.

1.2 Paper Organization

Section 2 details our approach for information-flow tracking and describes the untrusted sandbox U. Section 3 describes benign sandbox B. Policy inference is described in Section 4, followed by a description of our implementation and evaluation (Section 5). Related work is discussed in Section 6, followed by concluding remarks in Section 7.

2. Containing Untrusted Processes

Our untrusted sandbox, illustrated in Figure 2, consists of a simple inner sandbox U_I based on OS-provided access control mechanisms, and an outer sandbox that is realized using a library U_L and a user-level helper process U_H.

The inner sandbox U_I enforces an isolation policy that limits untrusted processes so that they can only write untrusted files (Section 2.1). This strict policy, by itself, can cause many untrusted applications to fail. The transparency library U_L (Section 2.2) component of the outer sandbox masks these failures so that applications can continue to operate as if they were executing directly on the underlying OS. In particular, U_L remaps some of the failed requests (primarily, system calls) so that they would be permitted by U_I. In other cases, it forwards the request to the helper process U_H, which runs with the userid of a normal user, to carry out the request. The helper U_H uses a policy that is more permissive than the inner sandbox, but will still ensure information-flow based integrity.

In addition to modifying or relaying requests from untrusted processes, the transparency library U_L may also modify the responses returned to them in order to preserve their native behavior. We provide two examples of remapping/relaying to illustrate its benefit:

- When a benign application is run with untrusted inputs, it will execute as an untrusted process, and hence will not be able to update its preference files. To avoid application failures that may result due to this, U_L can redirect these accesses to untrusted private copies of such files.
- Untrusted applications will experience a failure when they attempt to create files in the home directory of a user u, since this directory is not writable by untrusted userids. In this case, U_L can forward the request to the helper process, which runs with the privileges of u and hence can perform this access.

Whether a particular file access is remapped/relayed is determined by security policies, a topic further discussed in Section 4. Similarly, the policy enforced by U_H is also discussed below.

2.1 Inner Sandbox U_I

Contemporary desktop OSes provide access control mechanisms for protecting system resources such as files and IPCs. Moreover, processes belonging to different users are isolated from each other. We repurpose this mechanism to realize the inner sandbox. Such repurposing would, in general, require some changes to file permissions, but our design was conceived to minimize such changes: our implementation on Ubuntu Linux required changing permissions on less than 60 files (Section 5). Moreover, it preserves all of the functionality relating to the ability of users to share access to files.

The basic idea is to run untrusted processes with newly-created userids that have very little, if any, direct access to modify the file system. For each non-root userid[1] R in the original system, we add a corresponding untrusted userid R_u. Similarly, for each existing group G, we create an untrusted group G_u that consists of all userids in G and their corresponding untrusted userids. To further limit accesses of R_u, we introduce a new group G_b of existing ("benign") userids on the system before untrusted userids are added. File permissions are modified so that world-writable files and directories become group-writable[2] by G_b. Similarly, world-executable *setuid* programs are made group executable by G_b.

With the above permission settings, no R_u will have the permission to create files, and hence will need to rely on the helper process U_H to create them. Since U_H runs with the userid R, these files will be owned by R. To identify them an untrusted, U_H sets up the group owner of this file to be G_u, where G is the primary group of R. As a result, untrusted processes will not be able to change permissions on these files or overwrite them without the help of U_H, thus enabling the helper to exert full control over their access. Untrusted processes cannot modify benign files either, since the benign sandbox ensures appropriate permission settings on them.

Untrusted processes can compromise benign processes through communication. Some communication mechanisms, such as pipes between parent and child processes, need to be closed when a child process of a benign process becomes untrusted. This can happen in our system only through the `execve` system call. Other communication mechanisms such as signals and IPC are restricted by the OS based on userids, and hence the inner sandbox will already prevent them. For intra-host socket communication, the benign sandbox is responsible for identifying the userid of the peer process and blocking the communication. To block communication with external hosts, appropriate firewall rules can be used.

Using userid as an isolation mechanism has been demonstrated in systems like Android and Plash [2] for isolating applications.

[1] We don't support untrusted code execution with root privileges.

[2] If group permissions are already used, then we use ACLs instead.

One of our contributions is to develop a more general design that not only supports strict isolation between applications, but also permits controlled interactions. (Although Android can support interactions between applications, such interactions can compromise security, providing a mechanism for a malicious application to compromise another benign application. In contrast, our approach ensures that malicious applications cannot compromise benign processes.) Our second contribution is that our approach requires no modifications to (untrusted or benign) applications, whereas Android and Plash require applications to be rewritten so that they do not violate the strict isolation policy.

2.2 Transparency Library U_L

For untrusted processes, U_L replaces the standard C-library in our system, and provides its function through system call wrappers. Note that U_L operates with the same privileges as the untrusted process, so no special security mechanisms are needed to protect it. U_L's main purpose is to mimics unprotected execution environment for untrusted processes.

Userid and group transparency.

Applications may fail simply because they are being run with different user and group ids. For this reason, U_L wraps `getuid`-related system calls to return R for processes owned by R_u. It also wraps `getgid`-related system calls to return G for processes group-owned by G_u. This mapping is applied to all types of userids, including effective, real and saved userids. As a result, an untrusted process is not even aware that it is being executed with a different userid from that of the user invoking it.

This modification is important for applications that query their own user or groupid, and use them to determine certain accesses, e.g., if they can create a file in a directory owned by R. If not, the application may refuse to proceed further, thus becoming unusable. Some common applications such as `OpenOffice`, `gedit`, `eclipse` and `gimp` make use of their userid information. U_L ensures that such applications remain usable.

File access transparency.

When a file request is denied by the inner sandbox, U_L forwards the call transparently to the helper process U_H running with the privileges of R. U_H, if it chooses to permit the operation, will open the file and transmit the file descriptor back to U_L via a UNIX-domain socket. U_L then forwards this descriptor to the untrusted process. This technique enables subsequent read/write operations to be performed directly by the untrusted process, thereby avoiding a hop to U_H for most operations.

2.3 Helper Process U_H

In the absence of our protections, programs will be executed with the userid R of the user running it. Thus, the maximum access they expect is that of R, and hence U_H can be run with R's privileges.

Observe that the inner sandbox imposes restrictions (on R_u relative to R) for only three categories of operations[3]: file/IPC operations, signaling operations (e.g., `kill`), and tracing operations (e.g., `ptrace`). We have not found useful cases where R_u needs to signal or trace a process owned by R. IPC objects with permission settings are treated the same as files. Consequently, we focus the discussion on file system operations:

- *Reading user-readable files:* U_H permits an untrusted process owned by R_u to read any file that is readable by R, including files that do not have explicit read permission for R_u.
- *Executing user-executable files:* Except for setuid files, U_H permits an untrusted process with userid R_u to execute any file that can be executed by R.
- *Creating new files or directories in user-writable directories:* An untrusted process is permitted by U_H to create new files or directories in any directory writable by R.
- *Overwriting of existing files:* U_H permits any file overwrite that would succeed for R. However, unless the target file is untrusted, the original file is left unchanged. Instead, U_H transparently creates a private copy of the file for any subsequent use by R_u. File removals are treated in a similar way.
- *Operations to manipulate permissions, links, etc.:* These operations are handled similar to file modification operations: if the target file(s) involved is untrusted, then U_H permits the change but with integrity labels preserved. Otherwise, the changes are performed on a private copy of the original file that is created for R_u. As before, all references to the original file by R_u are redirected to this copy.

Note that redirection leads to namespace fragmentation: a file being accessed needs to be searched within the redirection space, and then the main file system. Users may have a hard time locating such files, as they are visible only to untrusted processes. Our implementation reduces this fragmentation by limiting redirection to application preference files: applications need to modify these files but users are unlikely to look for (or miss) them. Data files are not held in the redirection space. We discuss in Section 4.1 how to distinguish between these file types.

While we do not emphasize confidentiality protection, our system provides the basis for sound enforcement of confidentiality restrictions by tightening the policy on user-readable files.

3. Protecting Benign Processes

Our benign sandbox completes the second half of our sandbox architecture. Whereas the untrusted sandbox prevents untrusted processes from directly damaging benign files and processes, the benign sandbox is responsible for protecting benign applications from indirect attacks that take place through input files or interprocess communication.

A simple way to protect benign applications is to prevent them from ever coming into contact with any thing untrusted. However, total separation would preclude common usage scenarios such as the use of benign applications (or libraries) in untrusted code, or the use of untrusted applications to examine or analyze benign data. In order to support these usage scenarios, we partition the interaction scenarios into three categories as follows.

- *Logical isolation:* By default, benign applications are isolated from untrusted components by the benign sandbox.
- *Unrestricted interaction:* The other extreme is to permit benign applications to interact freely with untrusted components. This interaction is rendered secure by running benign applications within the untrusted sandbox.
- *Controlled interaction:* Between the two extremes, benign applications may be permitted to interact with untrusted processes while remaining a benign process. Since malware can exploit vulnerabilities of benign software through these interactions, they should be limited to *trusted* programs that can protect themselves in such interactions.

The first and third interaction modes are supported by a benign

[3]Recall that R cannot be `root`, and hence many system calls (e.g., changing userid, mounting file systems, binding to low-numbered sockets, and performing most system administrative operations) are already inaccessible to R-processes. This is why it is sufficient to consider these three categories.

sandboxing library B_L. As described Section 3.1, it enforces policies to protect benign code from accidental exposure to untrusted components. The second interaction mode makes use of the untrusted sandbox described earlier, as well as a benign sandboxing component (Section 3.2) for secure context switch from benign to untrusted execution mode.

3.1 Benign Sandboxing Library

Since benign processes are non-malicious, they can be sandboxed using a replacement library B_L for the standard C library. In the isolation mode, B_L enforces the following policies.

- *Querying file attributes:* Operations such as `access` and `stat` that refer to untrusted files are denied. An error is returned to indicate permission denial.
- `execve` *and* `open` *for reading:* These are handled in the same way as file attribute query operations.
- *Changing file permissions:* These operations are intercepted to ensure that benign files aren't made writable to untrusted users, and that untrusted files aren't turned into benign ones. These restrictions prevent unintended changes to the integrity labels of files. However, there may be instances where a benign process output needs to be marked untrusted. An explicit function is provided in the replacement C-library for this purpose.
- *Interprocess communication channel establishment:* This includes operations such as `connect` and `accept`. The OS is queried for the userid of the peer process. If it is untrusted, the communication will be closed and return a failure code.
- *Loading kernel modules:* If the OS provides a system call to load a kernel module using a file path, the library will deny this call if the file is untrusted. Otherwise, loading a module would require a process to `mmap` the module into its memory. Since this file open will be denied for untrusted files, they can't be loaded as kernel modules.

In addition to isolation, B_L can also support controlled interaction between benign and untrusted processes. This option should be exercised only with *trustworthy* programs that are designed to protect themselves from malicious inputs. Moreover, *trust should be as narrowly confined as possible,* so B_L can limit these interactions to specific interfaces and inputs on which a benign application is trusted to perform sufficient input validation.

B_L provides two ways by which trust-confined execution can deviate from the above default isolation policy. In the first way, an externally specified policy identifies the set of files (or communication end points such as port numbers) from which untrusted inputs can be safely consumed. The policies can also specify if certain outputs should be marked as untrusted. In the second way, a trusted process uses an API provided by B_L to explicitly bypass the default isolation policy, e.g., `trust_open` to open an input file even though it is untrusted. While this option requires changes to the trusted program, it has the advantage of allowing its programmer to determine whether sufficient input validation has been performed to warrant trusting a certain input.

3.2 Secure Context Switching

Switching security contexts (from untrusted to benign or vice-versa) is an error-prone task. One of the advantages of our design is that it leverages a well-studied solution to this problem, specifically, secure execution of setuid executables in UNIX.

A switch from untrusted to benign domain can happen through any setuid application that is executable by untrusted users. Well-written setuid programs protect themselves from malicious users. Moreover, OSes incorporate several features for protecting setuid executables from subversion attacks during loading and initialization. While these should be sufficient for a safe switching out of untrusted domain, our design further reduces the risk with a default policy that prevents untrusted processes from executing setuid executables. This policy can be relaxed for specific setuid applications that are deemed to protect themselves adequately.

Transitions in the opposite direction (i.e., from benign to untrusted) require more care because processes in untrusted context cannot be expected to safeguard system security. We therefore introduce a gateway application called `uudo` to perform the switch safely. Since the switch would require changing to an untrusted userid, `uudo` needs to be a setuid-to-root executable. It provides an interface similar to the familiar `sudo`[4] program on UNIX systems — it interprets its first argument as the name of a command to run, and the rest of the arguments as parameters to this command. By default, `uudo` closes all benign files that are opened in write mode, as well as IPC channels. These measures are necessary since all policy enforcement takes place at the time of `open`, which, in this case, happened in the benign context. Next, `uudo` changes its group to G_u and userid to R_u, and executes the specified command. (Here, R represents the real userid of the `uudo` process.)

We view `uudo` as a system utility, similar to `sudo`, that enables users to explicitly execute commands in untrusted mode. While it may seem like a burden to have to use it every time an untrusted execution is involved, experience with the use of `sudo` suggests that it is easy to get used to. Moreover, the use of `uudo` can be inferred (Section 4.2) in common usage scenarios: launching an application by double-clicking on a file icon, running an untrusted executable, or running a benign command with untrusted file argument.

4. Policy Inference

In the preceding sections, our focus was on policy enforcement mechanisms, and the different ways they could handle a particular access request. To build a practical system that preserves user experience, we need as much (if not more) emphasis on the policies that specify the particular way each and every request is handled. This is the topic of this section.

4.1 Untrusted Code Policy

Our policy for untrusted processes is geared to stop actions that have a high likelihood of damaging benign processes. A benign process may be compromised by altering its code, configuration, preference or input files. Of these, the first three choices have a much higher likelihood of causing harm than the last. For this reason, our policy for untrusted processes is based on denying access to code, configuration and preference files of benign processes. However, note that benign applications may be run as untrusted processes, and in this case, they may fail if they aren't permitted to update their preference files. For this reason, preference file accesses need to be redirected, while denying writes of configuration and code files.

To implement this policy, we could require system administrator (or OS distributors) to specify code, configuration and preference files for each application. But this is a tedious and error-prone task. Moreover, these details may change across different software versions, or simply due to differences in installation or other options.

A second alternative is to do away with this distinction between different types of files, and apply redirection to all benign files that are opened for writing by untrusted processes. But this approach

[4]The name `uudo` parallels `sudo`, and stands for "untrusted user do," i.e., execute a command as an untrusted user.

has several drawbacks as well:

- Redirection should be applied to as few files as possible, as users are unaware of these files. In particular, if data files are redirected, users may not be able to locate them. Thus, it is preferable to apply redirection selectively to preference files.
- If accesses to all benign files are redirected, this will enable a malicious application to compromise *all* untrusted executions of benign applications. As a result, no benign application can be relied on to provide its intended function in untrusted executions. (Benign executions are not compromised.)
- Finally, it is helpful to identify and flag accesses that are potentially indicative of malware. This helps prompt detection and/or removal of malware from the system.

We therefore develop an automated approach for inferring different categories of files so that we can apply redirection to a narrow subset of files.

Explicitly specified versus implicit access to files.

When an application accesses a file f, if this access was triggered by how it was invoked or used, then this access is considered to be explicitly specified. For instance, f may be specified as a command-line argument or as an environment variable. Alternatively, f may have been selected by a user using a file selection widget. A file access is implicit if it is not explicitly specified.

Applications seldom rely on an explicit specification of their code, configuration and preference files. Libraries required are identified and loaded automatically without a need for listing them by users. Similarly, applications tend to "know" their configuration and preference files without requiring user input. In contrast, data files are typically specified explicitly. Based on this observation, we devise an approach to infer implicit accesses made by *benign* applications. These accesses are monitored continuously, and a database of implicitly accessed files, together with the mode of access (i.e., read-only or read/write) is maintained for each executable. The policy for untrusted sandbox is developed from this information, as shown in Figure 3.

Note that our inference is based on accesses of benign processes. Untrusted executions (even of benign applications) are not considered, thus avoiding attacks on the inference procedure.

Computing Implicitly Accessed Files.

Files that are implicitly accessed by an application are identified by exclusion: they are the set of files accessed by the application but are not explicitly specified. Identifying explicitly specified files can be posed as a taint-tracking problem. Taint sources include: (a) command-line parameters, (b) environment variables, and (c) file names returned by a file selection widget. In addition to propagating taint in the usual way, i.e., through assignments and memory copying operations, there are a few other rules for propagation:

- if a file with a tainted name is opened, then all of the contents of the file are marked tainted.
- if a directory with a tainted file name is opened, then all of the file names from this directory are marked as tainted.

Finally, explicitly specified file names as those that are tainted.

In terms of implementation, we rely on taint inference [21] rather than taint analysis. Some aspects of the structure of file names are exploited to increase accuracy, and to deal with differences in the manner of specification of file names.

We construct a data structure to store the list of explicitly identified names for each process. These names are then matched against every open system call to identify explicitly accessed files. Because there can be multiple explicitly identified names, we used Aho-Corasick algorithm [4] for efficient string matching.

	Implicitly accessed by benign		Explicitly accessed
	read and write	other	
Inferred type	Preference	Code and configuration	Data
Action	Redirect	Deny	Deny

Figure 3: Untrusted Sandbox policy on writing benign files

4.2 Benign Code Policy

Policies can also be inferred for benign programs, although some of the aspects are too complex to resolve entirely automatically.

Logical isolation.

The default policy for benign code is to prevent consumption of untrusted inputs, while returning a "permission denied" error code.

Untrusted execution.

Requiring users to explicitly invoke `uudo` has the benefit that users know in advance whether they can trust the outputs or not. However, it is an inconvenience for users to make this decision all the time. Hence, our system can also automatically infer the use of `uudo`. The idea is as follows: if an execution will fail without `uudo` but may succeed with it, we automatically invoke `uudo`. Currently, we have implemented this for the simple cases of benign applications invoked with untrusted input files. This technique works well when applications are launched by a file manager when the user double-clicks a file, or uses a "open with" dialog. It also works for simple commands that take a file name argument. Handling more general cases, e.g., pipelines, is a topic of future work.

Trust-confined execution.

There does not seem to be a practical way to automatically decide which applications are *trustworthy.* However, it is possible to identify where trust is inappropriate: given the critical role played by implicitly accessed files, it does not seem appropriate to trust applications to defend themselves from untrusted data in these files. In this manner, the inference procedure described earlier is helpful for determining trust confinement policies.

5. Implementation and Evaluation

Our primary implementation was performed on Ubuntu 10.04. Fifteen assembly instructions were inserted around each system call invocation sites in system libraries (`libc` and `libpthread`). This allows us to intercept all system calls. Our implementation then modifies the behavior of these system calls as needed to realize the sandboxes described in Section 2 and 3. We also modified the loader to refuse loading untrusted libraries for benign processes.

When our system is installed, existing files are considered as benign. We found no world-writable regular files, so no permission changes were needed for them. There were 26 world-writable devices, but we did not change their permissions because they do not behave like files. We also left permissions on sockets unchanged because some OSes ignore their permissions. Instead, we perform checking within the `accept` system call. World-writable directory with sticky-bit set were left unmodified because OSes enforce a policy that closely matches our requirement. Half of the 48 world-executable setuid programs were modified to group-executable by G_b. The rest were setgid programs and were protected using ACLs.

There are a few pivotal benign applications such as web browsers, email readers and word processors that are exposed to a wide range of inputs. One way to use them safely is to run them as benign or untrusted process, based on the integrity of the input files. This works well for applications such as editors or document viewers.

	Shared	Ubuntu	PCBSD
Require no instrumentation	118	170	205
Benign Sandbox	49	6	29
Untrusted Sandbox	55	7	40

Figure 4: Number of system calls

However, some applications need to simultaneously process messages from benign and untrusted sources, e.g., browsers and file utilities. We have experimented with two approaches for such applications: (a) expect the application to protect its integrity from certain untrusted inputs, thus allowing it to have unrestricted interactions on those specific interfaces, and (b) use separate instances of the application when interacting with untrusted or benign data. We experimented with both choices for `Firefox` and `Thunderbird`. Many file utilities (`mv`, `cp`, `tar`, `find`, `grep`, and `rm`) represent mature programs, so we used option (a).

A key requirement for using option (a) is that applications need to label their outputs accordingly, instead of always labeling them as benign. For most file utilities, this is done by using appropriate flags. For `Firefox` and `Thunderbird`, we developed add-ons for this purpose.

Installation of untrusted software represents another key challenge, as administrative privileges are needed during installation, yet many components executed at install time are from untrusted sources. To address this challenge, we have developed an approach based on SSI [24] to secure this phase.

We also limited the privileges of untrusted X-clients with X-security extensions or nested X-server to protect other benign clients.

5.1 Portability and Complexity

To further establish the simplicity and practicality of our approach, we ported our system to PCBSD (version 8.2), one of the best known desktop versions of BSD. Similar to the implementation on Ubuntu, we modified the library by inserting assembly instructions at each system call invocation site.

Figure 4 shows the number of system calls we instrumented to enforce policies. On i386 Linux, some calls are multiplexed using a single system call number (e.g., `socketcall`). We demultiplexed them so that the results are comparable to BSD. Most of the system calls require no instrumentation. A large number of system calls that require instrumentation are shared between the OSes. Note that some calls, e.g., `open`, need to be instrumented in both sandboxes.

A large portion of the PCBSD specific system calls are never invoked: e.g., NFS, access control list, and mandatory access control related calls. Of those 59 (10 overlaps in both sandboxes) system calls that require instrumentation, 29 are in the benign sandbox. However, only 4 (`nmount`, `kldload`, `fexecve`, `eaccess`) out of the 29 calls are actually used in our system. Hence, we only handle these 4 calls. For the rest of the calls, we warn about the missing implementation if there is any invocation. The other 40 calls in untrusted sandbox are for providing transparency. We found that implementing only a subset of them (`futimes`, `lchmod`, `lutimes`) is sufficient for the OS and applications like `Firefox` and `OpenOffice` to run. Note that incomplete implementation in the transparency library U_L does not compromise security.

Figure 5 shows the code size for different components for supporting Ubuntu, and the additional code for PCBSD. The overall size of code is not very large. Moreover, a significant fraction of the code is targeted at application transparency. We estimate that the code that is truly relevant for security is less than half of that shown, and hence the additions introduced to the TCB size are modest. At the same time, our system *reduces* the size of the TCB by a much larger amount, because many programs that needed to be trusted to be free of vulnerabilities don't have to be trusted any more.

	LOC				
	C		header		Other
	Ubuntu	+PCBSD	Ubuntu	+PCBSD	Both
Shared	2208	130	737	27	39
helper U_H	703	16	106		
uudo	68	52			
$B_L \cap U_L$	811	15	492	30	74
B_L only	451	67			
U_L only	944	81			
Total	5185	361	1335	57	113

Figure 5: Code complexity on Ubuntu and PCBSD

Document Readers	Adobe Reader, dhelp, dissy, dwdiff, evince, F-spot, FoxitReader, Geegle-gps, jparse, naturaldocs, nfoview, pdf2ps, webmagick
Games	asc, gbrainy, Kiki-the-nano-bot, luola, OpenTTD, SimuTrans, SuperTux, supertuxkart, Tumiki-fighters, wesnoth, xdemineur, xtux
Editor/ Office/ Document Processor	Audacity, Abiword, cdcover, eclipse, ewipe, gambas2, gedit, GIMP, Gnumeric, gwyddion, Inkscape, labplot, lyx, OpenOffice, Pitivi, pyroom, R Studio, scidavis, Scite, texmaker, tkgate, wxmaxima
Internet	cbm, evolution, dailystrips, Firefox, flickcurl, gnome-rdp, httrack, jdresolve, kadu, lynx, Opera, rdiff, scp, SeaMonkey, subdownloader, Thunderbird, Transmission, wbox, xchat
Media	aqualung, banshee, mplayer, rhythmbox, totem, vlc
Shell-like	bochs, csh, gnu-smalltalk, regina, swipl
Other	apoo, arbtt, cassbeam, clustalx, dvdrip, expect, gdpc, glaurung, googleearth, gpscorrelate-gui, grass, gscan2pdf, jpilot, kiki, otp, qmtest, symlinks, tar, tkdesk, treil, VisualBoyAdvance, w2do, wmmon, xeji, xtrkcad, z88

Figure 6: Software tested

5.2 Preserving Functionality of Code

We performed compatibility testing with about 100 applications shown in Figure 6. 70 of them were chosen randomly, the rest were hand-picked to include some widely used applications.

5.2.1 Benign mode

First, we installed all 100 packages as benign software. As expected, all of them worked perfectly when given benign inputs.

To use these applications with untrusted inputs, we first ran them with an explicit `uudo` command. In this mode, they all worked as expected. When used in this mode, most applications modified their preference files, and our approach for redirecting them worked as expected.

We then used these applications with untrusted inputs, but without an explicit `uudo`. In this case, our `uudo` inference procedure was used, and it worked without a hitch when benign applications were started using a double-click or a "open-with" dialog on the file manager `nautilus`. The inference procedure also worked well with simple command-lines without pipelines and redirection. Further refinements to this procedure to handle pipelines and more complex commands is a topic of ongoing work.

5.2.2 Untrusted mode

We then configured the software installer to install these applications as untrusted. Remarkably, all of the packages shown in Figure 6 worked without any problems or perceptible differences. We discuss our experience further for each category shown in Figure 6.

Document Readers.

All of the document readers behave the same when they are used to view benign files. In addition, they can open untrusted files with-

out any issues. They can perform "save as" operations to create new files with untrusted label.

Games.

By default, we connect untrusted applications as untrusted X-clients, which are restricted from accessing some advanced features of the X-server such as the OpenGL GLX extensions. As a result, only 8 out of 12 games worked correctly in this mode. However, all 12 applications worked correctly when we used (the some what slower) approach of using a nested X-server (`Xephyr`).

Editors/Office/Document Processors.

These applications typically open files in read/write mode. However, since our system does not permit untrusted processes to modify benign files, attempts to open benign files would be denied. Most applications handle this denial gracefully: they open the file in read-only mode, with an appropriate message to the user, or prompt the user to create a writable copy before editing it.

Internet.

This category includes web browsers, email clients, instant messengers, file transfer tools, remote desktop clients, and information retrieval applications. All these applications worked well when run as untrusted processes. Files downloaded by applications are correctly labeled as untrusted. Any application opening these downloaded files will hence be run in untrusted mode, ensuring that they cannot damage system integrity.

Media Player.

These are music or video players. Their functions are similar to document readers, i.e., they open their input files in read-only mode. Hence, they do not experience any security violations.

Shell-like application.

This category includes shells or program interpreters that can be executed interactively like a shell. Once started in untrusted mode, all the subsequent program executions will automatically be performed in untrusted mode.

Other Programs.

We tested a system resource monitor (`wmmon`), file manager (`tkdesk`), some personal assistant applications (`jpilot`, `w2do`, `arbtt`), `googleearth` and some other applications. We also tested a number of specialized applications: molecular dynamic simulation (`gdpc`), DNA sequence alignment (`clustalx`), antenna ray tracing (`cassbeam`), program testing (`qmtest`, `expect`), computer-aided design (`xtrkcad`) and an x86 emulator (`bochs`). While we are not confident that we have fully explored all the features of these applications, we did observe the same behavior in our tests in benign as well as untrusted modes. The only problem experienced was with the application `gpscorrelate-gui`, which did not handle permission denial (to write a benign file) gracefully, and crashed.

5.3 Experience with Malicious Software

Here we illustrate scenarios involving stealthy attacks that are stopped by our system.

Real world malware.

Malware can enter systems during installation of untrusted software or via data downloads. As secure installation is not our focus, we assumed that attacks during installation are prevented by systems like [24] and untrusted files are labeled properly.

We tested our system with malware available on [1]. These malware were mainly rootkits: patched system utilities like `ps` and `ls`, kernel modules, and `LD_PRELOAD` based libraries. Specific packages tested include: `JynxKit`, `ark`, `BalaurRootkit`, `Dica`, and `Flea`. All of them tried to overwrite benign (indeed, root-owned) files, and were hence stopped.

`KBeast` (Kernel Beast) requires tricking root process to load a kernel module. The benign sandbox prevents root processes from loading the kernel module since the module is labeled as untrusted.

Real world exploit.

We tested an Adobe Flash Player exploit (CVE-2008-5499) which allows remote attackers to execute arbitrary code via a crafted SWF file. If the browser is simply trusted to be free of vulnerabilities, then this attacks would obviously succeed. Our approach was based on treating the web-site as untrusted, and opening it using an untrusted instance of the browser. In this case, the payload may execute, but its actions are contained by the untrusted sandbox. In particular, it cannot damage system integrity.

Simulated Targeted Attacks.

We also simulated a targeted attack via compromising a document viewer. A user received a targeted attack email from an attacker, which contained a PDF that can compromise the viewer. When the user downloaded the file, the email client labeled the attachment as untrusted automatically since the sender cannot be verified. Our system, however, did not prevent the user from using the document. User could still save the file along with other files.

When she opened the file, the document viewer got compromised. On an unprotected system, the attacker controlled viewer then dumped a hidden malicious library and modified the `.bashrc` file to setup environment variable LD_PRELOAD such that the malicious library would be injected into all processes the user invoked from shell. Worse, if the user has administrative privileges, the viewer can also create an alias on `sudo`, such that a rootkit would be installed silently when user performs an administrative action.

Although the viewer still got compromised on our system, the user was not inconvenienced: while she could view the document, modification attempts on `.bashrc` were denied, and hence malware attempts to subvert and/or infect the system were thwarted.

5.4 Performance

	Benign		Untrusted	
	Overhead	σ	Overhead	σ
`openssl`	0.01%	1.43%	-0.06%	0.70%
`Firefox`	2.61%	4.57%	4.42%	5.14%

Figure 7: Runtime overhead for Firefox and OpenSSL.

Figure 7 shows the overhead of `openssl` and `Firefox` when compared with unprotected systems. We obtained the statistics using `speed` option in `openssl`. As for `Firefox`, we used `pageloader` addon to measure the page load time. Pages from top 1200 Alexa sites were fetched locally such that overheads due to networking is eliminated. The overhead on `openssl` benchmark is negligible. The average overhead for `Firefox` is less than 5%.

Figure 8 shows the `SPEC2006` benchmark with the highest overheads. The overhead is less than 1% for CPU intensive operations.

Figure 9 shows the latency for some GUI programs. We mea-

	Unprotected	Benign	Untrusted
	Time (s)	Overhead	Overhead
403.gcc	541.2	-1.99%	0.82%
456.hmmer	982.7	0.36%	-0.13%
458.sjeng	933.8	0.49%	0.51%
462.libquantum	995.4	-0.17%	0.33%
433.milc	882.5	0.85%	-2.66%
Average		-0.10%	-0.28%

Figure 8: Highest 5 overhead in SPEC2006, `ref` input size

	Unprotected	Benign	Untrusted
	Time (s)	Overhead	Overhead
eclipse	6.16	1.99%	10.23%
evolution	2.44	2.44%	5.04%
F-spot	1.61	2.11%	6.80%
Firefox	1.32	3.24%	10.08%
gedit	0.82	5.02%	6.09%
gimp	3.63	1.90%	4.32%
soffice	1.56	0.33%	7.08%

Figure 9: Latency for starting and closing GUI programs

sured the time between starting and closing the applications without using them.

6. Related Work

System-call interposition and sandboxing.

Two of the most popular mechanisms for secure policy enforcement are Linux Security Modules (LSM) [27] and ptrace [19]. The drawbacks of kernel-based approaches (e.g., LSM) have been eloquently argued [12, 10]: kernel programming is more difficult, leads to less portable code, and creates deployment challenges. Approaches such as ptrace avoid these drawbacks by enabling policy enforcement to be performed in a user-level monitoring process. However, it poses performance problems due to the frequent context switches between the monitored and monitoring processes. More importantly, TOCTTOU attacks are difficult to prevent [9].

Ostia [10] avoided most of these drawbacks by developing a *delegating architecture* for system-call interposition. It used a small kernel module that permits a subset of "safe" system calls (such as `read` and `write`) for monitored processes, and forwards the remaining calls to a user-level process. Our system's use of a user-level helper process was inspired by Ostia. While their approach still requires kernel modifications, our design is implemented entirely at user-level by repurposing user access control mechanisms.

While many techniques have been focused on the mechanisms for confinement, the problem of developing effective policies has not received as much attention. Some works such as SELinux [17], Systrace [20] and AppArmor [6] were focused on protecting benign code, and typically rely on a training phase to create a policy. Such training-based approach is inappropriate for untrusted code. So Mapbox [3] develops policies based on expected functionality by dividing applications into various classes. Model-carrying code [22] provides a framework in which code producers and code consumers can effectively collaborate to come up with good policies. While it represents a significant advance over purely manual development of policies, it still does not scale to large numbers of applications. Supporting entire OS distributions, such as the work presented in this paper, would require a very large amount of effort.

Both our system and Plash [2] confine untrusted programs by executing them with a userid that has limited accesses in the system. Additional accesses are granted by a helper process. However, our focus is on providing compatibility with a wide range of software, while protecting the integrity of benign processes. We achieve this goal by systematically sandboxing all code, whereas Plash sandboxes only untrusted code with least privilege policies.

Isolation-based Approaches.

Applying two-way isolation for desktop OSes is particularly challenging because of how applications interact. Fragmented namespace, and excessive efforts needed to maintain multiple working environments make two-way isolation less attractive.

In contrast, two-way isolation is particularly popular for app model (e.g., Windows 8, Mac OS X, iOS, and Android) because apps only require limited interactions. Android relies on user permission to achieve two-way isolation, and this has some similarity with our reliance on user permissions to realize the inner sandbox. A difference is that the Android model introduces a new user for each application, whereas we introduce a new (untrusted) user for each existing user. Another important difference between the app model and our approaches is that in the apps world, composition of applications is the exception, whereas in our system, it is the norm. While the app models protect malicious code from subverting other apps directly, they do not protect against malicious data. Once data sharing takes place, there is no more security guarantees. We allow safe interactions to take place by running benign applications inside untrusted sandbox.

One-way isolation techniques, exemplified by Alcatraz [15], enforces a single, simple policy on all applications: they are permitted to read any thing on the system, but their effects are contained within an isolated environment. This simplifies the maintenance of the isolated environment. However, the approach has two significant drawbacks. First, if the results of isolated execution need to be used, it needs to be brought out of isolation, at which the system is potentially exposed to malware attacks. Second, almost none of the actions of untrusted code are denied by Alcatraz. This can be exploited by malware to quickly compromise all applications running in isolation, thus making the environment less than useful.

Information flow techniques.

Our approach can be regarded as an instance of classical information flow [8, 5], with group ownership standing for integrity labels. The closest to our work is PPI [25]: Our formulation of integrity is similar, both approaches are designed to provide integrity by design, and both approaches focus on automating policies. But there are several important advances we make in this work over PPI. First, we provide a portable implementation that has no kernel component, whereas the bulk of PPI resides in the kernel. Second, PPI approach for policy inference requires exhaustive training, the absence of which can lead to failures of benign processes. Specifically, incomplete training can lead to a situation where a critical benign process is unable to execute because some of its inputs have become untrusted. The approach presented in this paper avoids this problem by preventing any benign file from being overwritten with untrusted content. On the other hand, PPI provides some features that we don't: the ability to run untrusted applications with root privilege and dynamic context switch from high to low integrity. We do not provide these features because they do significantly complicate system design and implementation.

UMIP [14] focuses on protecting against network attackers. Unlike UMIP, which uses the sticky bit to encode untrusted data, our approach repurposes DAC permission to allow us to track untrusted data. Furthermore, in the desktop context, compromise of user files is an important avenue for malware propagation, but UMIP does not attempt to protect the integrity of user files. IFEDAC [18] extends UMIP to protect against untrusted users as well. Both UMIP and IFEDAC require additional code in kernel to enforce policy. Our approach avoids the need for kernel code. Another difference is that we sandox both benign and untrusted processes while they do not constrain benign processes. Specifically, it is possible in IFEDAC for a benign process to be accidentally downgraded due to the consumption of untrusted input, and this can cause all its future accesses to be denied, including writes to files that were opened before consuming untrusted input. Our approach avoids this *self-revocation problem* [8].

Another set of works focus on Decentralized Information Flow Control (DIFC) [13, 28, 7]. Instead of centrally specifying what is high integrity, DIFC allows applications to create their own in-

tegrity levels. As compared to our approach, DIFC approaches enable the enforcement of more expressive and flexible policies. Their downside is that they require nontrivial changes to the OS and/or applications to achieve security benefits, whereas our emphasis is on avoiding any changes to the OS and application code, while still achieving robust defense from malware.

7. Summary and Conclusions

We presented a new approach that provides principled protection from malware attacks: as long as untrusted content isn't mislabeled as benign, malware attacks are stopped, regardless of malware sophistication or the skills of its developers. Through experimental results, we showed that our approach achieves strong protection without significantly impacting the usability of benign and untrusted applications. To achieve this, we developed a novel dual sandboxing architecture that decomposes policies into two parts, one that is enforced on untrusted processes, and another on benign processes. A minimal policy is used to confine untrusted processes, making untrusted processes more usable. This policy is complemented by the policy enforced on benign applications. The two policies work together to provide strong separation between benign and untrusted contexts.

We also presented detailed policies that are enforced by each sandbox, and an inference procedure that serves to automate the identification of which policies are to be applied to which files. Our implementation has been greatly simplified by a design that achieves most enforcement in a cooperative setting with the processes on which the policies are being enforced. This enabled our our implementation to be compact, as well as portable. Our system introduces low performance overheads. An open-source software implementation of our system is available on the web [26].

8. References

[1] Packet storm, `http://packetstormsecurity.com`.

[2] Plash, `http://plash.beasts.org/contents.html`.

[3] Anurag Acharya, Mandar Raje, and Ar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Applications. In *USENIX Security*, 2000.

[4] Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. In *Commun. ACM 18(6)*, 1975.

[5] 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.

[6] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious Server Security. In *LISA*, 2000.

[7] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. In *SOSP*, 2005.

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

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

[10] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*, 2004.

[11] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In *USENIX Security*, 1996.

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

[13] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In *SOSP*, 2007.

[14] Ninghui Li, Ziqing Mao, and Hong Chen. Usable Mandatory Integrity Protection for Operating Systems . In *S&P*, 2007.

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

[16] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX ATC*, 2001.

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

[18] Ziqing Mao, Ninghui Li, Hong Chen, and Xuxian Jiang. Combining Discretionary Policy with Mandatory Information Flow in Operating Systems. In *TISSEC 14(3)*, 2011.

[19] Pradeep Padala. Playing with ptrace, Part I, `www.linuxjournal.com/article/6100`.

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

[21] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *NDSS*, 2009.

[22] R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. In *SOSP*, 2003.

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

[24] Weiqing Sun, R. Sekar, Zhenkai Liang, and V. N. Venkatakrishnan. Expanding Malware Defense by Securing Software Installations. In *DIMVA*, 2008.

[25] Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *S&P*, 2008.

[26] Wai Kit Sze. Portable Integrity Protection System (PIP). `http://www.seclab.cs.sunysb.edu/seclab/pip`.

[27] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *USENIX Security*, 2002.

[28] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *OSDI*, 2006.