Expanding Malware Defense by Securing Software Installations*

Weiqing Sun[1], R. Sekar[1], Zhenkai Liang[2] and V.N. Venkatakrishnan[3]

[1] Department of Computer Science, Stony Brook University
[2] Department of Computer Science, Carnegie Mellon University
[3] Department of Computer Science, University of Illinois, Chicago.

Abstract. Software installation provides an attractive entry vector for malware: since installations are performed with administrator privileges, malware can easily get the enhanced level of access needed to install backdoors, spyware, rootkits, or "bot" software, and to hide these installations from users. Previous research has been focused mainly on securing the execution phase of untrusted software, while largely ignoring the safety of installations. Even security-enhanced operating systems such as SELinux and Vista don't usually impose restrictions during software installs, expecting the system administrator to "know what she is doing." This paper addresses this "gap in armor" by securing software installations. Our technique can support a diversity of package managers and software installers. It is based on a framework that simplifies the development and enforcement of policies that govern safety of installations. We present a simple policy that can be used to prevent untrusted software from modifying any of the files used by benign software packages, thus blocking the most common mechanism used by malware to ensure that it is run automatically after each system reboot. While the scope of our technique is limited to the installation phase, it can be easily combined with approaches for secure execution, e.g., by ensuring that all future runs of an untrusted package will take place within an administrator-specified sandbox. Our experimental evaluation has considered over one hundred benign and untrusted software packages. Our technique was able to block malicious packages among these without breaking non-malicious ones.

Keywords: Untrusted code, Malicious code, Software installation, Sandboxing.

1 Introduction

Malware, including adware, spyware, rootkits, backdoors, trojans, and bot software, has become a major security concern on desktop systems over the past few years. Although it was common in the past for software to be executed automatically when users click on attachments or hyperlinks, this practice is no longer that common: execution of untrusted software[4] typically requires explicit user consent, or an exploit on web browser or email handler[5].

Software installation provides a more attractive entry vector for malware than competing alternatives such as remote exploits since installations are usually carried out with highest (administrative level) privileges, thereby providing malware the level of access it needs to embed itself deeply and firmly into the system, and to hide its pres-

* This research is supported in part by an ONR grant N000140710928 and NSF grants CNS-0627687, CNS-0716584 and CNS-0551660.

[4] We use the term "untrusted software" to refer to software obtained from untrusted sources on the Internet. Untrusted software may be malicious or non-malicious. On the other hand, benign software, which is obtained from trusted sources, is assumed to be non-malicious.

[5] This observation is supported by a white-paper from Symantec [11], which indicates that most adware and spyware enter desktop systems via an explicit software installation step.

ence from system monitoring utilities. In contrast, programs targeted by exploits (including those embedded in e-mail attachments or browser links) may run with user-level privileges, making it harder for malware to embed itself into the system. Furthermore, security-conscious users can deploy defenses against remote exploits (by using firewalls, buffer overflow defenses such as address-space randomization, etc.) and malicious e-mail attachments and other implicitly downloaded programs (by automatically sandboxing them). In contrast, few defenses are available to secure software installations. Even secure operating systems such as SELinux don't usually impose restrictions during software installs, expecting the system administrator to "know what she is doing." Unfortunately, even the most sophisticated users typically do not understand what goes on when complex software packages are installed. Often, these packages run scripts or other programs with administrative privileges, with the user having no knowledge of these activities. Software installations thus provide an ideal vehicle for malware to surreptitiously inject itself into a host system.

In spite of the threats posed by the installation phase, previous research on untrusted software security [17,4,27,31,22,37,33] has been focused primarily on their execution phase. Relatively little work has been done on securing the installation phase. This paper seeks to address this overlooked problem, and develops a solution that works well with existing techniques for securing the execution phase. Specifically, our technique achieves the following goals that we consider essential for secure software installation:

- *Untrusted software should not interfere with the operation of benign packages.* Sophisticated spyware and rootkits can hide themselves in such a way that trusted components in the system end up executing their malicious payload. Since trusted system components aren't typically sandboxed or carefully monitored, this makes it easier for malware to execute without being noticed.
- *Untrusted software should not be allowed to execute outside a user-specified sandbox or virtualization environment.* Some malware may cause damage during installation, but others may cause damage when they are run. To guard against the latter, our approach can install untrusted code in a manner that it cannot be run outside a sandbox (or a virtualization environment).
- *Untrusted software should be (securely) uninstallable at any time.* Malware may install itself in such a way that uninstallation won't work properly. For instance, they may use scripts to copy files that are unspecified in the package; these files won't be removed during uninstallation.

Our technique does not make many assumptions about what constitutes a software installation — it may involve running a software package manager such as RedHat's `rpm`, Debian's `dpkg`, running a self-installing executable, installation from a tarball, etc. It may also involve running higher-level GUI-based installers that in turn invoke these lower level installation mechanisms. Our key observation is that once there is an explicit user consent involved, at that point, we can "wrap" the command that is executed for installation so that it runs within our Secure Software Installer (*SSI*).

The rest of this paper is organized as follows. Section 2 describes our threat model. Section 3 presents an overview of our approach and describes the high level design of Secure Software Installer (*SSI*). Section 4 describes the installation policies implemented in *SSI*. Section 5 presents an experimental evaluation of *SSI*. Related work is discussed in Section 6, followed by concluding remarks in Section 7.

2 Threat Model and Defense Overview

Our approach is based on the availability of mechanisms to distinguish between benign and untrusted software. For instance, all software that is digitally signed by a trusted vendor may be classified as benign, while the rest may be deemed untrusted.

We divide the threats posed by untrusted software into three phases: installation phase, execution phase, and uninstallation phase. A variety of solutions are available for securing the execution phase, including runtime policy enforcement (also known as sandboxing) [27,23,17,31,6,4,25,12], isolated execution [22,32,37], and file-label-based integrity protection [33,28]. Therefore, this paper is concerned only with the installation and uninstallation phases. Nevertheless, to demonstrate the end-to-end feasibility of our approach, our implementation includes a defense for the execution phase.

Software installation (and uninstallation) requires a higher level of privilege and access than the execution phase. This makes it difficult to define policies that ensure security objectives without breaking installations. The central contribution of this paper is that of developing policies and enforcement techniques to overcome this challenge.

2.1 Install-time Threats

We assume that *the goal of malware is to execute some or all of its code while being free of the above-mentioned confinement mechanisms* that are to be employed during the execution phase of untrusted software. Before enumerating possible ways in which this goal can be achieved, it is helpful to have an understanding of the main features of modern software package managers. The specific details given here pertain to RedHat Package Manager (RPM), although the description is applicable (with minor changes) to other package managers such as Debian's `dpkg`.

An RPM package contains a dozen or more components, most of which are descriptive in nature, e.g., name, version, vendor, copyright, URL, etc. There are five components that are security-relevant:

- *Files contained in the package,* i.e., the files copied by RPM during installation.
- *Scripts.* A package may contain several shell scripts that are run at various stages of installation such as before building a source package, before installation, after installation, etc. RPM runs these scripts at the specified stage.
- *Requires.* This tag specifies dependencies that a package may have. A package may depend on one or more packages. Rather than specifying these dependencies using package names, RPM and Debian permit the use of arbitrary strings. A package that has a dependency s will be installed only if there is already another package installed on the system that "provides" s. The use of arbitrary strings for dependencies allows for multiple implementations of the same functionality.
- *Provides.* The functionality provided by a package. It will be matched against the "requires" field as described above.
- *Conflicts.* If a package conflicts with one or more packages, those are listed in this section. A new package that conflicts with an existing package will not be installed.

Based on the above description, the following attack avenues are possible that may let untrusted code to escape confinement:

1. *Attacks that perform malicious actions at install time.* RPM does not pose any restrictions on the scripts contained in a package. Thus, in the absence of additional

protection, arbitrary attacks on the host are possible. *SSI* performs the installation within a virtual environment, so that these attacks would be isolated from the host.

The only way in which the host environment is affected in *SSI* is due to copying of files modified during the installation — these files are copied out of the virtual environment onto the host. Hence the rest of the discussion below is concerned with how files may be used to achieve the goals of malicious code.

2. *Attacks that modify files used by benign packages.* By modifying these files, a malicious package may be able to inject its code into the execution flow of a benign application. Since benign applications are not constrained in any way, such an attack would allow malicious packages to escape confinement. There are two cases to consider here:
   - *Existing benign packages.* A malicious package may claim to contain a library or executable that is already used by an existing benign application. As a result, these files may be overwritten when the package is installed, and hence future runs of this benign application may end up executing code that belongs to an untrusted package. It is also conceivable that an attack based on modifying a non-code file (e.g., configuration file used by a benign application) may subvert the operation of a benign application and cause it to execute the code of an untrusted package. *SSI* prevents these attacks by restricting untrusted packages from modifying (or deleting) any existing file other than those previously installed by an untrusted package.
   - *Benign packages installed in the future.* Instead of targeting an existing benign application, a malicious package may target a package that is likely to be installed in the future. Alternatively, it may claim to provide (in the sense of "provides" feature described above) a functionality needed by a future benign package. In these cases, *SSI* would permit the initial installation of these files belonging to the untrusted application. However, at the time of installation of the benign package, *SSI* will detect that a benign package depends on an untrusted package, or contains files belonging to an untrusted package. In either case, *SSI* disallows installation and notifies the user so that he/she can uninstall the untrusted package before attempting to install the benign package (possibly after installing additional benign packages that satisfy the dependencies of the current benign package).

   The above discussion assumes that package dependency information is complete. However, it is possible that some optional libraries or configuration files may be omitted in the package specification. Worse, for software installed from tarballs, no dependency information is available. *SSI* employs a second line of defense to prevent untrusted libraries and executables from being directly used by benign applications. It installs libraries in separate directories that are included in the search path used by the dynamic loader for untrusted applications, but not for benign applications[6]. Untrusted executables are installed in such a way that when they are invoked, they are run within a confinement environment.

   While our approach copes with missing dependencies on library or executable files, it does not currently implement a complete defense against missing configuration file dependencies. This is partly because we considered it a low-risk, and partly be-

[6] On Linux, this is done by including these directories in the `LD_LIBRARY_PATH` environment variable before running an untrusted application, and not including them for benign applications.

cause the threat could be eliminated in the isolation-based execution confinement mechanism used in our implementation. However, as described in Section 5.1, our experiments suggest that a more general solution would be based on restricting the data files written by untrusted applications.

3. *Attacks contained in the files belonging to an untrusted package.* As described above, *SSI* ensures that all executables belonging to the untrusted code are "wrapped" in such a manner that when they are invoked, they would automatically be started up within a sandbox or virtual environment.
4. *Attacks on integrity of package database.* Package managers typically use a few files to maintain a database of packages installed on the system. Since many of the policies described above were based on the content of this database, these policies can be undermined by attacks that compromise the integrity of the database. To preclude these attacks, *SSI* verifies that the database changes resulting from the installation of an untrusted package concern that package only, and do not modify (or insert) information about other packages.

Our discussion in this paper is focused primarily on integrity threats, and does not consider denial-of-service threats[7].

2.2 Uninstall-time Threats

Software uninstallation is carried out with the same level of privileges as the installation phase. Contemporary package managers run scripts provided by the package. Thus, the threat model parallels that of the installation phase. Specifically, it consists of:

1. *Attacks that perform malicious actions during uninstallation.* These remain the same as during installation, and are addressed in the same way.
2. *Attacks that leave behind files after uninstallation.* We do not distinguish in this case between different types of files, or whether these files relate to benign packages in any way. Instead, *SSI* ensures that all files that were installed by an untrusted package are removed on uninstallation.
3. *Attacks that remove files belonging to other packages. SSI* enforces a policy that ensures that only the files copied into the host at installation time can be removed at uninstall-time.
4. *Attacks on the integrity of package database.* The attacks discussed in the installation phase under this category continue to be possible at uninstallation time, and can be prevented using the same high level policies (i.e., ensuring that the database updates are consistent with the package removed.)
5. *Attacks that cause errors during uninstall.* Such attacks are possible if the scripts related to the package perform actions that lead to an error, which in turn cause the package manager to abort uninstallation. While errors would cause a rollback during the installation phase, it is not an option here: we wish to remove the package. Our approach is to use the "force" option provided by package managers to forcibly remove the package from the database. (As mentioned above, *SSI* already ensures that the files installed by the package are removed.)

[7] This is why "conflicts" did not enter the discussion above — a malicious package may claim to conflict with a large number of packages that are likely to be installed in the future. When a user attempts to install them, she will get an error message. It is expected that in this case, and in other cases involving conflicts or failures relating to untrusted packages, the user will uninstall the untrusted package before proceeding further.

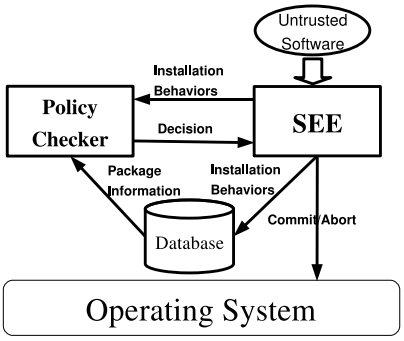

Fig. 1. Design and operation of *SSI*.

3 Approach Overview

Our approach consists of the following phases:

- *Initial installation in a virtual environment,* where the installation can proceed without violating host integrity or install-time failures. The actions observed during the installation are logged for further analysis in the policy-checking phase.
- *Policy checking* to detect if the actions observed during initial installation violated the requirements captured by an installation policy.
- *Commit/abort* phase, which propagates the files modified during installation to the host if no policy violations occurred. Otherwise, installation is aborted, leaving the host state as if the installation never took place.
- *Secure execution* phase, during which untrusted software can be invoked within a confinement mechanism that is specified at install time.
- *Secure uninstallation* phase that ensures that untrusted software can be uninstalled safely at any time.

These phases (and their rationale) are described in more detail below. Figure 1 shows the components of *SSI* involved in the installation as well as the uninstallation phase.

3.1 Initial Installation Phase

There are two basic options for protection against attacks during the installation phase. First, the installation could be performed within a sandbox that prohibits the execution of any action that has the potential to compromise host security. Unfortunately, such an *eager enforcement* approach is likely to fail: software installation typically requires writing to system directories, and updating databases that record the software installed on the system. Denying these actions will lead most installations to fail, while permitting them has the potential to damage system integrity. In particular, there is no easy way to determine whether an individual database update is safe or not: it is the end result achieved by a series of updates that can be determined to be "safe" or "unsafe." For this reason, *SSI* determines safety by first performing the installation within a virtual environment, and examining post-installation system state for verifying safety policies. As we describe later, such state-based policies provide a novel capability that is crucial for expressing and enforcing the safety requirements for securing software installations.

We rely on our Safe Execution Environment (SEE) [32] for initial installation. SEEs offer several benefits over the alternative of using virtual machines for this purpose.

Chief among them is that of accurate environment reproduction: SEEs are based on one-way isolation, which makes the host state visible inside the SEE. In other words, they provide an initial environment that is exactly the same as the host environment. As such, software installations, which have a number of host dependencies (including those based on previously installed software, their releases and patch versions, and so on) can be successfully installed within the SEE if they can be installed on the host OS. In contrast, virtual machines require significant additional effort for exact duplication of the host environment.

The second important reason for using SEE is that they offer the ability to commit the results of installation onto the host environment. If we relied on virtual machines, there is typically no easy way to migrate the changes made within the VM to the host OS. The obvious approach of rerunning the installation on the host OS after policy verification can turn out to be dangerous: a malicious software package may detect that it is being run within a VM the first time, and may not exhibit malicious behavior. For this reason, our installation policy may hold for the installation within the VM. However, when the installation is rerun on the host, malicious software can detect that it is no longer within a VM, and exhibit malicious behavior that violates our policy. In contrast, with the SEE, the behavior verified against a safety policy is the same one that gets committed to the host, thus ensuring that installation policies cannot be violated.

Our approach can support software installation using means other than package managers, e.g., tarballs and and self-installing executables. This is because our approach has no direct dependency on the tools used for installation — they are simply run inside the SEE, and the resource accesses are observed, and policies enforced on their basis.

SSI uses the Alcatraz tool [22,5] for realizing an SEE. Alcatraz uses copy-on-write to handle file operations, i.e., any host files modified within the SEE are copied into the SEE and modified. The modifications are not visible to host processes unless they are also running within the same SEE. Modifications involving other resources (e.g., mounting files, arbitrary communication with processes outside SEE) are controlled by a policy that forbids most accesses that have the potential to harm host security. More details on SEE implementation (including the containment policies used) can be found in [22,32]. For *SSI*, we made a few modifications to Alcatraz: (a) replacement of manual determination of safety with an automated policy enforcement mechanism, (b) support for the Secure File Container feature described later, and (c) selective relaxation of restrictions on non-file resource accesses within Alcatraz so that software installers can download software from the Internet.

3.2 Policy Checking Phase

Previous work on SEE relied on a manual approach for determining the safety of the actions performed by untrusted software. Unfortunately, such a manual approach is cumbersome and error-prone. We have therefore developed an automated approach for determining the safety of software installations. Safety is defined by a policy, which is derived from the high-level description provided in Section 2. An important innovation in our approach is the development and use of *state-based policies* that can refer to the operations performed during installation, as well as the *actual end result* of installation. Such state-based policies are strictly more powerful than the class of policies that are enforceable using runtime monitoring [29], where decisions regarding permissibility

of an operation need to be made without knowing about future operations made by a program. For instance, an installation program may need to add a new userid to the password file, and may do this by creating a copy of the password file, editing it to add a user, removing the original password file and then renaming the copy. A runtime monitoring approach would have to prevent the removal step of the password file, whereas a state-based policy can check that the end result of the program is acceptable: specifically, the difference between the initial and final password file is the addition of a line that corresponds to the new user, respecting other criteria such as the use of previously unused user and groupids, and a permitted shell.

A second innovation in our policies is that of action attribution: instead of requiring policies to be specified entirely in terms of low-level operations (or state changes), our policy framework allows these low-level operations to be mapped to higher-level operations, and the specification of policies in terms of these operations. Taking the userid addition example again, rather than stating a policy that relies on computing file differences between the original and modified password files and verifying certain characteristics of these differences, we can instead correlate the changes to the execution of a program *useradd:* in this case, the policy can be simpler, stating that the execution of `useradd` command with certain arguments is permitted.

Different policies can be associated with different installations — our policy framework provides flexibility in this regard. However, in practice, we expect a small number of policies to be sufficient. One policy would concern benign packages, while a small set of policies may be specified for untrusted packages. (Our implementation uses a single policy for all untrusted applications, although this will need to be changed if we wish to support untrusted applications that require a higher level of access, e.g., servers that get started automatically after reboot.) The specific policies used in our implementation are described in Section 4.

Package Database. The policy checker makes policy decisions by querying a database, which consists of two components:

- Package management database. It is used by an existing package manager such as RPM or Debian to store information about the contents and dependency of all the installed packages.
- *SSI*-database. It is used to maintain package names, trust labels, and information about software installed outside of a package manager, such as tarballs and self-installing executables.

SSI currently supports RPM database, but its dependence on the details of the database is minimal. The implementation needs to be able to query RPM about the packages installed on the system, and their dependencies. For this reason, *SSI* can be easily ported to other package managers such as Debian. Moreover, *SSI* has no dependency on the higher level tools used during installation, e.g., Gnorpm or Synaptic package manager. These tools are simply run inside the SEE, and the safety policies checked against the resulting system state and the actions observed within the SEE.

3.3 Commit/Abort phase

If the policy checker reports success, then the results of installation are committed. Otherwise, the entire SEE is discarded, which ensures that the host OS state is unchanged by the installation phase. The commit/abort phase is provided by SEE: we made one

change, as described below, to ensure that untrusted software would always execute within a user-specified sandbox.

3.4 Secure Execution of Installed Software

An untrusted application may not violate any install-time policy, but may still exhibit malicious behavior when it is run. For instance, a game program may also act as a "bot," polling an attacker-specified network address for malicious actions to carry out. Or, it may communicate with benign processes and may attempt to compromise them. For these reasons, it is important that the untrusted code be monitored at runtime, and its actions confined to ensure that it cannot compromise system security. We consider three options in this regard: sandboxing, isolated execution, and OS-based integrity protection.

Sandboxing. A number of sandboxing and policy confinement techniques have been developed [27,23,17,31,6,4,25,12], and may be used with *SSI*. *SSI* relies on a simple technique to ensure sandboxing of untrusted executables: while copying an executable from the SEE to the host OS, it is renamed, and the execution permission is removed. Libraries used by untrusted applications are stored in non-standard directories to minimize the likelihood that they could be accidentally used by benign applications. A wrapper script is created with the original name of the executable, which is then responsible for properly setting up the search path used by the dynamic loader, and executing the original executable within the sandbox. Note that this simple approach can be defeated by the user, but this is not our concern since we assume that the user is cooperative, i.e., the user will not actively subvert *SSI*.

Development of suitable sandboxing policies is a research problem in itself, and is not the focus of this paper. We simply observe that sandboxing policies are relatively easy to develop for some classes of untrusted code that are most commonly used, namely, document viewers and media players, as they require minimal access to OS resources.

Isolation. Instead of using a sandbox, the execution phase may rely on an isolation-based approach. This is the easiest option in our implementation since we are already using an isolation based technique in *SSI*. To ensure isolated execution of untrusted code, we modified Alcatraz so that it commits the results of untrusted installations to a separate section of the filesystem called a Secure File Container (SFC). The use of SFC ensures that none of the files (libraries, executables, or configuration files) contained in the untrusted software package can be accidentally used by benign applications. We use the same technique as with the sandboxing approach for invoking untrusted executables: a wrapper script is created with the original name of such executables. This wrapper script starts Alcatraz, initializes it with the environment within the SFC, and starts execution of the original executable.

As in the case of sandboxing, there remain some usability issues with isolation-based techniques — this is a topic of ongoing research in safe execution of untrusted software. As advances are made in this area, they can be seamlessly integrated with our approach focused on secure installations.

Information-Flow Based Integrity Protection. *SSI* will work seamlessly with information-flow based integrity techniques for Linux [33,28,21]. Indeed, *SSI* has been developed so that, together with the PPI integrity technique described in [33], it can provide

a comprehensive defense against malware. In particular, *SSI* can simply label the files belonging to untrusted application with low integrity, while files belonging to benign packages are labelled with high integrity. Since PPI ensures that information cannot flow from low-integrity sources to high-integrity sinks, it makes sure that benign processes and the files used by them won't be corrupted by untrusted applications.

3.5 Secure Uninstallation phase

Secure uninstallation is supported for untrusted packages. If a package A is to be uninstalled, we go ahead and uninstall all other packages that depend on A. Since our policies do not permit benign packages to depend on untrusted packages, uninstallation of untrusted packages can always be performed without breaking benign packages.

The threats relating to uninstallation phase and the approach for mitigating them were already discussed in Section 2, while the specifics of our policy are described in Section 4.2.

Within *SSI*, uninstallation first runs the normal package uninstallation process (e.g., `rpm -e`). It then determines if the actions performed during the uninstallation are permitted by the uninstallation policy specified in Section 4.2. Otherwise, *SSI* forces the package manager to remove the package from its database (without actual uninstallation), and then deletes all the files installed by the untrusted package.

4 Installation Policies

4.1 Policy Framework

One of the main difficulties with policy-based approaches is the difficulty of policy development. Sandboxing policies can routinely get quite large and complex since (a) they are stated in terms of low-level primitives (which files can be accessed and which ones can't be), and (b) there are a large number of files on the system, and it is time-consuming (and error-prone) to identify all files that an application should be permitted to access. Moreover, a different policy is needed for each application, as the set of allowable and/or required resource accesses differ for different applications.

We observed that the principal reason for policy complexity is the large gap between high-level policy objectives such as those stated in the Introduction, and the low-level policies that can actually be enforced, which deal with specific resources that can be accessed, and the operations that are permissible. To combat this problem, we developed an approach that enables automated generation of lower-level policies from higher level policies. The specific techniques and mechanisms used to support higher level policies are described below.

Deriving low-level, enforceable policies from software package dependencies. We leverage the contents of software packages to ensure that untrusted packages cannot modify or corrupt files used by benign packages. Specifically, the following pieces of information can be obtained from a software package: (a) the files contained in the package, and (b) the names of other packages that this package depends on. The second type of information is readily available for RPM or Debian packages, but not for tarballs or self-installing executables. This has not been a serious problem in practice since we need (b) only for benign packages, which are typically from an OS distribution vendor that uses a package manager such as RPM or Debian. However, if it becomes

necessary to install a benign package that arrives in the form of a tarball, the following work-around could be used to obtain an approximation for dependency information. In particular, the application can be executed within a virtual environment (e.g., our SEE) and its file accesses observed. The application then has a dependency on all the packages that contain one or more of the files accessed by the application. We note that the list obtained in this way may not be complete, but is clearly an improvement over the alternative of assuming no dependencies. Moreover, as described in Section 2, our approach incorporates a second line of defense to guard against attacks that may be possible due to incomplete dependence information.

To use the above procedure, benign packages need to be identified. We expect this information to remain the same across a given OS version, although it is conceivable that individual users[8] may have some differences in terms of the sources they are willing to trust. Such differences may be captured by appropriately modifying a configuration file that specifies this information.

In our implementation, where RPM is the default package manager, we query the installed packages on the system, and based on the signature of the RPM package, a trust label is assigned and recorded in the *SSI*-database. We verify that installed benign packages only depend on other benign packages. (If this is not true, there is an inconsistency, and user input is needed to resolve it.) For packages that are installed outside of the package manager, their contents (and optionally, dependencies) are maintained in the *SSI*-database.

State-based policies. Another important reason for the complexity of typical sandboxing policies is due to the need to ensure that each permitted action leaves the system in a safe state. This requires explicit consideration of all possible operations that can be performed by an application, and their possible operands, and identification of those operation/operand combinations that are safe. Since there can be many ways for an attack to achieve the same objective, the size (and complexity) of policies can correspondingly increase. Moreover, as illustrated using the user addition example earlier, some sequence of operations may first take the system to an unsafe state before bringing it back to a safe state.

For the reasons mentioned above, *SSI* uses state-based policies that can reference (a) the final state of the system, (b) the initial state of the system, and (c) the sequence of operations that took the system from the initial to the final state. This enables powerful policies to be specified, e.g., we can capture any sequence of operations that allow "a file f to be updated to an f' such that f and f' differ in at most k lines, and all these lines match a regular expression R."

The power offered by our post-execution analysis framework has steered us towards an extensible approach for verifying state-based policies, where new policy primitives could be defined by essentially writing scripts that operate on the state within the SEE, and return $true$ or $false$ indicating whether the policy was satisfied. We have chosen this alternative for expediency, as opposed to defining a special-purpose policy language.

Providing safe exceptions using action attribution. Sometimes, the installation of a package may require modifications to some files whose integrity is critical. For instance,

[8] Our intent is that the "user" is a system administrator — e.g., an OS distribution vendor may provide the list of benign and untrusted packages, or they may be maintained by user communities.

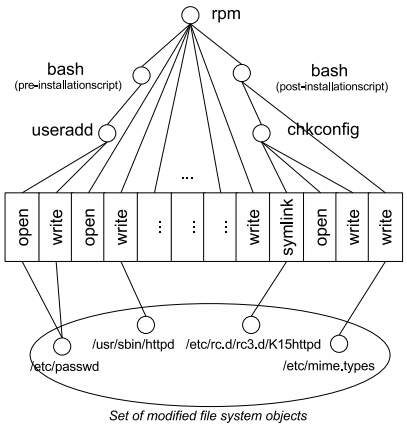

Fig. 2. Behavior of Apache Installation.

`/etc/ld.so.cache` file needs to be updated after installing new shared libraries. Similarly, some packages may need to create new users. Arbitrary changes to files such as `/etc/ld.so.cache` and `/etc/passwd` will harm the system, so *SSI* needs to provide mechanisms to perform controlled updates to these files that ensure safety.

One approach for permitting safe changes was described in the previous paragraph: by comparing modifications to the file, and defining safety criteria for these modifications. However, an alternative approach may be preferable in some cases. This approach exploits the fact that often, the system already provides utilities for safely updating certain critical pieces of information. Examples include the `ldconfig` program to update the `ld.so.cache` file, `useradd` and `groupadd` programs to create new users or groups, and `chkconfig` program to enable or disable automatic startup of a specified service.

Based on the above observation, our approach allows specification of policies that permit execution of such utility programs, with constraints on argument values. Such an approach avoids the need for writing policies that need to "understand" the format of configuration files. For instance, instead of describing the format of a "safe" entry in `/etc/passwd`, we can state that it is safe to call the `useradd` program with certain parameters, e.g., with a userid other than 0, and not belonging to any existing group.

Policies in terms of higher-level actions such as `useradd` are supported by the policy checker as follows. First, a raw log of operations performed within SEE is obtained. The policy checker analyzes this log to derive parent-child relationships between processes, the programs executed by each process, and the resource accesses made by them. This information can be represented using a tree structure shown in Figure 2. In this tree, the internal nodes represent processes, while the leaves represent modification operations. The program corresponding to the root process of this tree is `rpm`, as we used `rpm` from the command-line in this example.

If the policy states that `useradd` can be used with certain restrictions on parameters, the policy checker first verifies if the invocation of this program in the SEE log conforms to these restrictions. It also makes sure that the program did not interact with any untrusted components, other than being invoked from an untrusted script with the arguments as permitted by the policy. If these checks succeed, all operations in the log

that can be attributed to `useradd` or one of its children are deleted. Policies regarding resource accesses are checked after this step.

The power of the attribution mechanism is easier to illustrate in the context of more complex software packages. For this reason, Figure 2 shows the attribution tree for Apache. The scripts of this packages are executed in child processes of the `rpm` process. The pre-installation script is executed first, adding a user account though the `useradd` command. Then `rpm` copies the contents of the package into their destination. Finally, a post-installation script uses `chkconfig` to start up Apache automatically at boot time, and then updates `/etc/mime.types` file.

4.2 Policy for Installing Untrusted Packages

Our installation policy consists of the following components. These components correspond directly to the threat model described in Section 2. We also describe the enforcement of these policies based on the mechanisms and techniques described above.

We remark that the policy described below is exactly the same as the one used in our evaluation.

1. *Attacks that perform malicious actions at install time.* These are prevented by policies that are already enforced by SEE, which confine non-file accesses made within SEE. We made two modifications to the default policy: the installer application is permitted to access the network, so that it can download packages from the Internet if needed. We also make an exception for communication with the X-server. (Alternatively, untrusted applications may be directed to a nested X-server using the Xnest [3]. This option ensures that the primary X-server is not compromised by untrusted code.)
2. *Attacks that modify files used by benign packages.*
   - *Files that an existing benign package depends on. SSI* ensures that an untrusted package does not modify or delete any existing file, except possibly those installed by an untrusted package.
   - *Files that a future benign package depends on.* As mentioned earlier, this is prevented by enforcing a policy that restricts benign packages from (a) having dependencies on untrusted packages, and (b) containing files that belong to untrusted packages. The contents of the package manager database and *SSI* database are used to compute the complete list of files that are within a package, as well as the complete list of files that it depends on.
   - *Files used by a benign package without specifying dependency.* Since we do not know that such a file would be used by an existing or future package, no install-time policy can be specified to preclude this. Instead, this possibility is avoided at the time of execution of benign software. The exact mechanisms differ, depending upon the technique used during execution phase, and were described in Section 3.4.
3. *Attacks contained in files belonging to untrusted package.* These attacks are contained using a confinement mechanism during the execution of untrusted software. The choices for doing this were described in Section 3.4.
4. *Attacks on integrity of package database.* We enforce a policy that ensures that the changes to the package database are consistent with the files actually copied into the system. In particular, (a) the package contents should include all and only the files

that were reported as having been created or modified within the SEE, and (b) any files that were reported as having been deleted within the SEE must be part of the package. Moreover, information regarding all other packages in the database should remain unchanged.

5. *Granting exceptions based on attribution.* Updates to the file `/etc/ld.so.cache` using `ldconfig` are always permitted. Addition of a new MIME type in the file `/etc/mime.types` is permitted as long as it conforms to the state-based policy described before. These exceptions are recorded in *SSI* database so that their inverse operations can be permitted during uninstallation.

4.3 Policy for Uninstallation of Untrusted Packages

The uninstallation policy follows the outline specified in Section 2.

1. *Attacks that perform malicious actions during uninstallation.* These remain the same as during installation.
2. *Attacks that leave behind files after uninstallation.* The contents of the package database are queried to obtain the list of files installed by an untrusted package. If all these files are not removed during uninstallation, they are forcibly removed.
3. *Attacks that remove files belonging to other packages.* Once again, the contents of the package database are queried for the list of files installed by the untrusted application. Only these files are permitted to be deleted at commit time.
4. *Attacks on the integrity of package database.* These are thwarted by checking that the only change to the database is the removal of the untrusted package, and that none of the information relating to other packages have been changed.
5. *Attacks that cause errors during uninstall.* These attacks are handled as described in Section 3.5.
6. If *SSI* database indicates that exceptions were granted, operations that have the inverse effect are permitted.

Currently, there is no general way to identify how to "invert" an operation. Instead, we manually specify how to invert an operation on a case-by-case basis. For instance, for an operation that adds a MIME type, we specify that inverse operation has the effect of deleting the added MIME type.

4.4 Installation Policy for Benign Packages

The only policy enforced is that *benign packages should not depend on untrusted packages.* No policies are enforced during uninstallation of benign packages.

5 Evaluation

We have implemented *SSI* on RedHat Linux (CentOS 4.1). Our implementation uses a publicly available tool Alcatraz [5] as the SEE. The implementation of the policy checker and the user interface consists of 7K lines of Java code. In this section, we present an evaluation of the functionality and performance of this implementation.

5.1 Evaluation of Functionality

The goal of this section is to evaluate the utility of *SSI* in securing real-world software packages. In this regard, we considered four cases: (a) installation of malicious packages, (b) installation of nonmalicious untrusted packages, (c) installation of benign

packages, and (d) secure uninstallation. Of these, (a) and (b) use the policies described in Sections 4.2, (c) uses policies from Section 4.4, while (d) uses policies from Section 4.3.

Real-world packages don't embody all aspects of malicious behavior considered in Section 2. As a result, they do not stress our policies. In other words, confidence in the security provided by our approach is more a function of the completeness of the threat model and the soundness of the policies described earlier rather than the experimental evaluation. However, our experiments on nonmalicious and benign software involved a much larger number of packages, so they do demonstrate that our policies do not lead to false positives for typical untrusted packages.

Installation of Malicious Packages.

Ideally, *SSI* would be evaluated by experimenting with a large collection of malware samples. Unfortunately, such an evaluation is not feasible on our chosen platform (Linux) since malware is relatively uncommon on Linux. What we have been able to do is to evaluate *SSI* using rootkits that are available from [1] — these were the only malware collection that we were able to obtain. In addition, since these rootkits are easily detected by our technique, we developed two additional test cases that embody a more sophisticated attack strategy.

More generally, we observe that most malware is designed so that it runs in the background, and is started up automatically at boot time. This requires modification of startup files, e.g., files within `/etc/init.d/` on Linux. Since these files belong to benign packages, *SSI* will likely detect such attempts and abort the installation of such packages.

- *Disguised rootkit.* In this experiment, we downloaded all the rootkits that were available from [1]. There were a total of 10, of which 8 were applicable to Linux. Of these 8, four (`mood-nt`, `adore-ng`, `suckitdid` and `cd00r`) expect users to knowingly run them each time, and hence are not persistent. Our tool is not designed to prevent a knowledgeable user from knowingly running malware, but is rather aimed at malware that is installed surreptitiously. We were then left with four rootkits: `bobkit`, `tuxkit`, `lrk5` and `portacelo`. During installation, all these rootkits modified files belonging to benign packages, such as `ls`, `find`, `du`, `ps` and `init`. The installation analysis determined that these actions are in conflict with the security policy that only untrusted files can be overwritten by untrusted packages. Hence the installation was aborted cleanly.
- *Fake patch from Redhat.* We tried to install the patch for fileutils that was suggested in the phish email from Redhat [24]. This fake patch was stopped by *SSI*, as the installation policy identified that the patch tried to create a privileged user with no password. On seeing this violation, the installation was aborted.
- *"Malicious" rpm package.* The Fedora package build system [16] suggests three possible attack scenarios from the malicious package writer. Of these, a malicious rpm-scriptlet is a serious threat. To test the effectiveness of *SSI* under this threat, we crafted a "malicious" rpm package. This package is named `glibsys` in RPM format. During the installation phase, the package tried to overwrite system files `/lib/libc.so` and `/bin/gcc`. By running the installation inside *SSI*, the policy checker captured these unsafe behaviors and aborted the installation.

Installation of Nonmalicious Packages from Untrusted Sources.

For this test, we installed untrusted (but nonmalicious) packages from sources that might be considered untrustworthy, such as freshrpms and ATrpms. We report our experiences in installing and using these packages with *SSI*. In particular, we downloaded 335 packages from Atrpms and 152 packages from freshrpms. Only 144 of these 487 packages could be installed on our system *even in the absence of SSI* — this was because of dependencies that were not satisifed. Of these 144, 11 were server applications that required a higher level of trust, so we were left with 133 packages in all. Below are some examples of these applications.

- *Multimedia and Document Viewers:* gthumb, graphviz, ggv, xmms and xpdf.
- *Games, Web Agents, IM:* gnapster, ltris, xifrac, ymessenger and gaim.
- *Archive Creation and Related Utilities:* jpeg2ps, f2c, flac, unrar and pdfmerge.
- *File Organization and Album Creation:* hardlink++ and mkpp.
- *Editors:* bluefish, glabels, screem and gedit.

All of these 133 packages could be installed successfully without any problems within *SSI*. Thus, there were no false positives due to *SSI* in this experiment. Although we currently do not restrict the data files (e.g., configuration or documentation files) belonging to untrusted applications, we observed that we could do so fairly easily. In particular, we noticed that all these files had the name of the untrusted package, and were created within certain directories such as `/usr/share/doc` and `/usr/share/info/-nasm.info.gz`. There were about half-a-dozen such locations. Based on this observation, we plan to constrain data files written.

Installation of Benign Packages.

For this evaluation, we chose a set of 38 rpm packages from the official repository, and tried to install them within *SSI*. It turned out that 37 of them were installed successfully, and one of them (`ethereal`) complained that it was dependent on package `libnet` which was untrusted. On seeing this, we replaced the untrusted version of libnet with a benign version obtained from the official repository and repeated the installation process. During this second attempt, `ethereal` was installed without problems.

Secure Uninstallation.

We did not find any package that comes with malicious uninstallation scripts, so we hand-crafted some test cases to evalute the ability to perform secure uninstallation. In particular, we crafted a package which tried to delete `/etc/passwd` in its uninstallation script. This action was captured by *SSI* and it was a violation of the policy specified in Section 4.2. Therefore, this action was aborted, and *SSI* verified that the set of files installed were actually removed from the file system.

We then tried to uninstall the nonmalicious packages installed before. We randomly chose 10 of them and ran uninstallation operation within *SSI*, it turned out all of them were successfully uninstalled without any violations to the uninstallation policy.

5.2 Performance Evaluation

The result of performance evaluation is summarized in Table 1. We evaluated *SSI* using three types of installation packages: binary installer, tar ball distribution, and rpm distribution. Mozilla installer is a self-contained binary, and it performed 8716 file mod-

	Original Installation	*SSI* Installation	
	Time	Time	Overhead
Mozilla installer (binary)	3.285	4.127	26%
Gnuchess (tar ball)	15.868	18.98	20%
Yahoo! Messenger (rpm)	2.433	4.813	98%

Table 1. Performance overhead of *SSI*. All numbers are in seconds.

ifications using 6 child processes. It incurred an overhead of 26%. The installation of gnuchess package (tgz format) had a 20% overhead, and its operation included three steps: `configure`, `make`, and `make install`. The entire procedure involved creation of 1935 new processes and 5325 modification operations on the file system. Finally, the installation of Yahoo messenger (rpm package) forked 6 child process and involved overall 42650 modification operations on the file system, and it incurred a 98% overhead. The average overhead across these three packages is about 50%, which is moderate but we believe to be acceptable in the context of *SSI*. Moreover, the primary performance bottleneck is the Alcatraz tool that provides our SEE. It uses `ptrace`-based system call interception, which frequently introduces 100% overheads on programs.

To estimate the performance benefits achievable using a more efficient system call interposition mechanism, we made an enhancement to Alcatraz that uses in-kernel system call interception mechanism for operations that don't require processing by Alcatraz, e.g., read and write operations. With this modification, overheads due to context switches are decreased to about a third of the figures reported above. For instance, the overhead for installing Yahoo messenger rpm becomes 38% as compared to 98% which we observed using original Alcatraz.

6 Related Work

Software Installation. A number of recent research efforts have focused on the problem of software installation, but they are mainly concerned with handling dependencies and conflicts among packages.

Checkinstall [15] is a tool to build installation packages such as RPM from an installation script. Nix [14] presents a comprehensive solution for deploying software, but its focus is on functionality rather than security.

RPMShield [34] is a tool aimed at securing the process of software installation. It uses policies based on the notion of ownership of files by packages. A file is said to be owned by a package if it is part of that package. However, it does not address the problem of dependencies between benign and untrusted packages, nor does it satisfy any of the goals for secure software installation that we outlined in the Introduction.

SoftwarePot [20] incorporates a secure software circulation model for software deployment. The software to be run is encapsulated with a file system that is transferred from the code producer to consumer. The operations from the software are confined within the "pot." It can be thought of as a combination of sandbox and software distribution model. But it constrains users into using one single way of software installation and execution confinement method. As a result, it is not possible to utilize existing package formats or sandboxing tools and policies with this approach. In contrast, *SSI* is compatible with existing software installation methods, and it is flexible in allowing users to choose different execution confinement tools. More importantly, SoftwarePot

requires policy development efforts to support new software, while *SSI* uses a single installation policy for all untrusted applications. For securing untrusted applications during execution, *SSI* can leverage confinement policies that may already be available in widely used sandboxing tools such as systrace.

Virtualization and Isolation Approaches. Virtual Machines (VMs) [35,13,9] provide a coarse-granularity approach for dealing with untrusted software: such software could be run inside a virtual machine, while benign software runs on the host OS. FreeBSD jails [19], Linux VServer [2] and Solaris Zones [26] provide light-weight virtualization, where the same OS kernel is shared across the VMs, while still providing strong isolation between applications running on different VMs.

The main problem with virtualization approaches is that typically, users want to use untrusted software to operate on their files, and other resources that are part of the host OS. To derive the same utility within the VM, the host environment has to be duplicated inside the virtual machine. This is quite time-consuming — for instance, most standard (and typically benign) packages would have to be installed on both the host OS and the VM. Moreover, files needed by untrusted applications would need to be explicitly copied into the VM. As a result of this inconvenience, users frequently end up installing untrusted software directly on the host OS. Techniques such as Alcatraz [22] and FVM [37] (and the closely related product called Software Virtualization Solution (SVS) [7]) mitigate the overhead of environment duplication by using one-way isolation, wherein the host OS files are visible within the isolated environment, but the files written within the isolated environments aren't visible on the host OS. However, usability issues still remain: if users want to make use of the outputs produced by untrusted software, they have to explicitly copy them back into the host systems.

DTE and Sandboxing. Boebert and Kain proposed Domain and Type Enforcement (DTE) [10,36]. Subjects (processes) are associated with domains, while objects (e.g., files) are associated with types. DTE policies specify which domains can access which types. They also specify domain transitions (if any) that should take place when a certain program is executed. Use of DTE to defeat rootkit attacks is described in [8]. SELinux [23] security is primarily based on DTE policies that have been developed with the goal of enforcing the principle of least privilege.

A number of so-called "sandboxing" approaches that have been developed to address untrusted code security [17,12,4,25,30,27] are conceptually similar to DTE. Motivated by simplicity, many of these systems typically use policies that are based on program names and file names, eliminating the intermediary notions of types and domains. While this loses some generality, it seems acceptable in the context of untrusted software.

All of the above approaches can potentially be used during the resident phase of untrusted software. However, they do not provide the power or flexibility of *SSI* during the installation phase. First, all these techniques are only capable of enforcing safety properties [29], which require that every operation leaves the system in a "safe" state. As described in Section 3, software installations typically involve intermediate states that are not safe. This motivated the development of state-based policies in *SSI*. Second, one of the biggest challenges in using DTE and sandboxing techniques is the difficulty of policy development. In contrast, *SSI* uses a single high-level policy that is enforced on all untrusted package installations.

Information-flow based Approaches. There has been a resurgence of interest in information-flow based approaches for preserving host integrity. PPI [33] and SLIM [28] enforce information flow policies that ensure that high integrity objects and subjects are not compromised by interacting with low-integrity objects and subjects. While policy development has been a challenge that has impeded deployment of mandatory access control (MAC), recent efforts such as UMIP [21] and PPI [33] have begun to address this problem by developing techniques to synthesize MAC policies.

SSI complements the above techniques: while the above techniques can protect host integrity from the execution of untrusted software, they do not provide a good solution for the installation phase. However, they do provide a strong foundation for *SSI* since they can answer questions regarding the trustworthiness of every file on the system. As a result, some of the potential gaps in *SSI* policies that arise due to missing information in software packages can be avoided.

Back to the Future system [18] uses information flow techniques to detect the presence of malware. Their approach does not constrain malware during its installation; instead, it is detected when its files are used by a benign application. Its main advantage is that it can recognize any attempt by malware to inject itself into inputs consumed by benign applications. Its drawback is that it allows host integrity to be compromised (as a result of malware installation), and this change has to be undone when malware is detected. This rollback may cause delays, and moreover, can introduce subtle file system consistency issues.

7 Conclusion

Software installations provide an attractive avenue for spyware and rootkits to embed themselves deeply into the operating system. In this paper, we proposed an approach for securing this entry point by developing a framework that confines accesses made by untrusted packages during their installation. Our technique can support a diversity of software installation mechanisms. It can also work with different approaches for confining untrusted software after the installation phase. A key novelty in our approach is the development of a high-level policy framework that largely eliminates the need for developing application specific installation policies. Instead, a single, intuitively simple high level policy can be used for a wide range of untrusted applications. Our experimental results demonstrate that our approach is effective, and achieves the goals set out in the Introduction.

Acknowledgments

We thank Joy Dutta, Milan Manavat, and Kumar Thangavelu for their work on early versions of SSI, and Anupama Chandwani and Abhishek Dhamija for discussions on Redhat/Debian package managers. We also thank our shepherd John McHugh and the anonymous reviewers for their insightful comments and suggestions.

References

1. Linux rootkits. http://www.eviltime.com/download.php?page=hacking&subpage=rootkits.
2. Linux v-server. http://linux-vserver.org.
3. Xnest. http://www.xfree86.org/4.2.0/Xnest.1.html.
4. Anurag Acharya and Mandar Raje. Mapbox: Using parameterized behavior classes to confine applications. In *USENIX Security Symposium*, 2000.

5. Alcatraz. Available at download area of http://www.seclab.cs.sunysb.edu.
6. A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations, 1998.
7. Altiris. Software virtualization solution, 2005. http://www.altiris.com.
8. Lee Badger, Daniel F Sterne, David L Sherman, Kenneth M Walker, and Sheila A Haghighat. A domain and type enforcement unix prototype. In *USENIX Computing Systems*, pages 127–140, 1995.
9. Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating systems principles*, pages 164–177, 2003.
10. W E Boebert and R Y Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, pages 18–27, 1985.
11. Eric Chien. Techniques of adware and spyware. *Symantec*, April 2005.
12. A. Dan, A. Mohindra, R. Ramaswami, and D. Sitaram. Chakravyuha: A sandbox operating system for the controlled execution of alien code. Technical report, IBM T.J. Watson research center, 1997.
13. Jeff 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, October 10–14 2000.
14. Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A safe and policy-free system for software deployment. In *LISA*, pages 79–92, 2004.
15. Felipe Eduardo. Checkinstall, 2004. http://asic-linux.com.mx/˜izto/checkinstall/.
16. *The fedora.us buildsystem*. http://enrico-scholz.de/fedora.us-build/html/.
17. Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *USENIX Security Symposium*, 1996.
18. Francis Hsu, Thomas Ristenpart, and Hao Chen. Back to the future: A framework for automatic malware removal and system repair. In *Annual Computer Security Applications Conference (ACSAC)*, December 2006.
19. Poul H. Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, 2000.
20. Kazuhiko Kato and Yoshihiro Oyama. Softwarepot: An encapsulated transferable file system for secure software circulation. In *ISSS*, pages 112–132, 2002.
21. Ninghui Li, Ziqing Mao, and Hong Chen. Usable mandatory integrity protection for operating systems. In *IEEE Symposium on Security and Privacy*, 2007.
22. Zhenkai Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *ACSAC*, pages 182–191, 2003.
23. Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux o perating system. In *Proc. FREENIX track of the 2001 Usenix Annual Technical Conference*, 2001.
24. PHCN. Fedora-redhat fake security alert / trojan source code analysis, 2004. http://www.phcn.ws/main/include.php?path=content/articles.php&contentid=120&PHCN=.
25. Vassilis Prevelakis and Diomidis Spinellis. Sandboxing applications. In *Proceedings of Usenix Annual Technical Conference: FREENIX Track*, 2001.
26. Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *LISA*, pages 241–254. USENIX, 2004.
27. Niels Provos. Improving host security with system call policies. In *Proceedings of the 11th USENIX Security Symposium*, pages 257–272, 2003.
28. David Safford and Mimi Zohar. A trusted linux client (tlc), 2005.
29. Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
30. Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of Annual Computer Security Applications Conference*, 2002.
31. 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 *Proceedings of 19th ACM symposium of Operating Systems Principles (SOSP)*, Bolton Landing, New York, October 2003.
32. Weiqing Sun, Zhenkai Liang, V. N. Venkatakrishnan, and R. Sekar. One-way isolation: An effective approach for realizing safe execution environments. In *NDSS*, 2005.
33. Weiqing Sun, R. Sekar, Gaurav Poothia, and Tejas Karandikar. Practical proactive integrity preservation: A basis for malware defense. In *IEEE Symposium on Security and Privacy*, May 2008.
34. V. N. Venkatakrishnan, R. Sekar, T. Kamat, S. Tsipa, and Z. Liang. An approach for secure software installation. In *Proceedings of the 16th Systems Administration Conference (LISA-02)*, pages 219–226, Philadelphia, PA, November 3–8 2002.
35. Brian Walters. VMware virtual platform. *j-LINUX-J*, 63, July 1999.
36. W. D. Young, P. A. Telega, W. E. Boebert, and R. Y. Kain. A verified labeler for the Secure Ada Target. In *Proc. National Computer Security Conference*, pages 55–61, 1986.
37. Yang Yu, Fanglu Guo, Susanta Nanda, Lap chung Lam, and Tzi cker Chiueh. A feather-weight virtual machine for windows applications. In *Proceedings of the 2nd ACM/USENIX Conference on Virtual Execution Environments (VEE'06)*, June 2006.