Secure Isolation and Migration of Untrusted Legacy Applications

Shaya Potter Jason Nieh Dinesh Subhraveti
Computer Science Department
Columbia University
{spotter, nieh, dinesh}@cs.columbia.edu

Columbia University Technical Report CUCS-005-04, January 2004

Abstract

Existing applications often contain security holes that are not patched until after the system has already been compromised. Even when software updates are applied to address security issues, they often result in system services being unavailable for some time. To address these system security and availability issues, we have developed peas and pods. A pea provides a least privilege environment that can restrict processes to the minimal subset of system resources needed to run. This mechanism enables the creation of environments for privileged program execution that can help with intrusion prevention and containment. A pod provides a group of processes and associated users with a consistent, machine-independent virtualized environment. Pods are coupled with a novel checkpoint-restart mechanism which allows processes to be migrated across minor operating system kernel versions with different security patches. This mechanism allows system administrators the flexibility to patch their operating systems immediately without worrying over potential loss of data or needing to schedule system downtime. We have implemented peas and pods in Linux without requiring any application or operating system kernel changes. Our measurements on real world desktop and server applications demonstrate that peas and pods impose little overhead and enable secure isolation and migration of untrusted applications.

1 Introduction

As software complexity grows and computers become more interconnected, the need for effective computer security increases. Complex software often contains programming errors, some of which may lead to vulnerabilities that can be exploited by attackers who gain access to those applications. Standard security models employed by commodity operating systems, such as Unix, do not help this situation. Because Unix lumps all privileges together as root, an application that only periodically needs one privilege still needs to run as root, providing it with all privileges. An attacker can thus gain root privileges by exploiting a weakness in an application run as root. Consequently, Internet accessible services offer prime opportunities for remote attackers to gain access to applications running with privilege.

Security problems can wreak havoc on an organization's computing infrastructure. To prevent this, software vendors frequently release patches that can be applied to address security issues that have been discovered. However, software patches need to be applied to be effective. It is not uncommon for systems to continue running unpatched applications long after a security exploit has become well-known [35]. This is especially true of the growing number of server appliances intended for very low-maintenance operation by less skilled users. Furthermore, once a patch has been released, exploits of unpatched applications based on reverse engineering the patch now occur as quickly as a month later whereas such exploits took closer to a year just a couple years ago [23].

Software updates to existing applications may not address security problems that result from users accidentally downloading and executing malicious code. Recently a security hole was discovered in a popular mp3 player [19] that could result in arbitrary code being executed if a user played a maliciously constructed mp3. If the mp3 player were run within a simple sandbox that limited the player to one's collection of mp3s, the damage the malicious code could accomplish would be severely limited. Over the years, complex services like Sendmail have similarly been exploited to allow malicious code to be run within its context. Since Sendmail runs with privilege, the malicious code also runs with privilege. A sandbox can be used to protect an entire machine from a faulty service, such as Sendmail. However, these services don't run by themselves, but also depend on other aspects of the machine, such as programs a user might want to call from a Procmail script to filter their mail. Consequently, one might end up including the entire machine within the sandbox. Since common sandboxes simply provides a single namespace, they don't provide good security solutions for the complex services in use today.

Furthermore, even when software updates are applied to address security issues, they commonly result in system services being unavailable. Patching an operating system can result in the entire system having to be down for some period of time. If a system administrator chooses to fix an operating system security problem immediately, he risks upsetting his users because of loss of data. Therefore, a system administrator must schedule downtime in advance and in cooperation with all the users, leaving the computer vulnerable until repaired. If the operating system is patched successfully, the system downtime may be limited to just a few minutes during the reboot. If the patch is not successful, downtime can extend for many hours while the problem is diagnosed and a solution is found. For systems that need to provide a high degree of availability, downtime due to security-related issues is not only inconvenient but costly as well. While application servers can sometimes mirror application state between servers and allow an application to continue even when one server has to be taken down, they only work in specific situations. For instance, a regular user's desktop can not be mirrored between servers. Even for applications that can mirror their data, the application has to be designed to interface with the mirroring architecture, resulting in application specific solutions that are difficult to generalize.

We introduce Pea-Pods to provide a solution to these security problems. Pea-Pods provide two key abstractions, peas (Protection and Encapsulation Abstraction) and pods (PrOcess Domain). A pod is a lightweight migratable virtual execution environment that looks just like the underlying operating system environment. A pea is a least privilege environment within a pod that allows access to a subset of processes and resources in the pod. In tandem, peas and pods decouple process execution from the underlying operating system to provide transparent, secure isolation and migration of untrusted applications. Pea-Pods can isolate untrusted applications within sandboxes, preventing them from causing harm to the underlying system or other applications if they are compromised.

Pea-Pods can encapsulate a group of processes within a migratable sandbox environment that can be transparently moved from one machine to another, even when the systems are running different operating system versions with different security and maintenance patches. This enables security patches to be applied to operating systems in a timely manner with minimal impact on the availability of application services by migrating applications to another machine that has already been updated while the original system is brought down for security upgrades and maintenance. Once the original machine has been updated, applications can be migrated back and continue to execute even though the underlying operating system has changed. Pea-Pods provide migration using a checkpoint-restart mechanism that can also enable application services to be checkpointed before a system goes down and restarted when it comes back up. This provides fast recovery from system downtime even when other machines are not available to migrate application services, as well as providing a general solution that any application can take advantage of.

Pea-Pods achieve these goals through three distinguishing characteristics. First, a pod provides a consistent private virtual namespace that gives all processes within it the same virtualized view of the system. This virtualized view isolates sandboxed processes from the underlying system by associating virtual identifiers with operating system resources and only allowing access to resources that are made available within the virtualized namespace. This isolation mechanism provides a simple way to control what operating system resources are accessible to a group of processes. Similarly, it allows a pod to define a complete set of users which can be distinct from those supported by the underlying system.

Second, a pea provides a least privilege encapsulation layer within a pod that can limit certain processes from interacting with other processes and accessing file system and network resources. This is effective for preventing compromised applications from attacking other processes and resources of the system. We provide intuitive tools to easily and dynamically create Pea-Pods tailored for individual applications or groups of applications.

Third, Pea-Pod virtualization is integrated with a checkpoint-restart mechanism that decouples processes from dependencies on the underlying system and maintains process state semantics to enable processes to be migrated across different machines. The checkpoint-restart mechanism employs an intermediate format for saving the state associated with processes and Pea-Pod virtualization. This format provides a high degree of portability to support process migration across machines that are running operating systems that differ in the security and maintenance patches applied. It also enables application services to be checkpointed on a system and restarted after the underlying operating system is upgraded and the system is restarted.

We have implemented Pea-Pods in a prototype system as a loadable Linux kernel module. We have used this prototype to securely isolate and migrate a wide range of unmodified legacy and network applications. We measure the performance and demonstrate the utility of Pea-Pods across multiple systems running different Linux 2.4 kernel versions using three real-world application scenarios, including a full KDE desktop environment with a suite of desktop applications, an Apache/MySQL web server and database server environment, and a Sendmail/Procmail e-mail processing environment. Our performance results show that Pea-Pods can provide secure isolation and migration functionality on real world applications with low overhead.

This paper describes how Pea-Pods can isolate applications to limit their ability to attack a system and how Pea-Pods can migrate applications across operating system kernel changes to facilitate kernel maintenance and security updates with minimal application downtime. Section 2 describes the pea and pod abstractions in further detail. Section 3 presents the virtualization architecture to support the Pea-Pod model. Section 4 discusses the Pea-Pod checkpoint-restart mechanisms used to facilitate migration across operating system kernels that may differ in maintenance and security updates. Section 5 analyzes the security of Pea-Pods and illustrates the utility of the system in several application scenarios. Section 6 presents experimental results evaluating the overhead associated with Pea-Pods and measures the system performance in providing secure isolation and migration for several application scenarios. Section 7 discusses related work. Finally, we present some concluding remarks.

2 Pea-Pod Model

The Pea-Pod model provides two key abstractions, pods (PrOcess Domain) and peas (Protection and Encapsulation Abstraction). Pods enable secure isolation and migration of application components that only need to interact via the file system or Internet communication. Peas provide fine-grain isolation among application components that may need to interact using interprocess communication mechanisms, including signals, shared memory, IPC messages and semaphores, and process forking and execution.

A pod is a host-independent virtualized view of an operating system in which a group of processes can be executed. A pod may contain one or many processes, and a system may contain one or many pods. The pod abstraction provides the same application interface as the underlying operating system so that legacy applications can execute in the context of a pod without any modification. Processes within a pod can make use of all available operating system services, just like processes executing in a traditional operating system environment. Unlike a traditional operating system, the pod abstraction provides a self-contained unit that can be isolated from the system, checkpointed to secondary storage, migrated to another machine, and transparently restarted, as shown in Figure 1. This is made possible because each pod has its own private, virtual namespace. All operating system resources are only accessible to processes within a pod through the pod's private, virtual namespace.

A pod namespace is private in that only processes within the pod can see the namespace. It is private in that it masks out resources that are not contained within the pod, including processes outside of the pod. Processes inside a pod appear to one another as normal processes that can communicate using traditional IPC mechanisms. Other processes outside a pod do not appear in the namespace and are therefore not able to interact with processes inside a pod using IPC mechanisms such as shared memory and signals. Instead, processes outside the pod can only interact with processes inside the pod using network communication and shared files that are normally used to support process communication across machines.

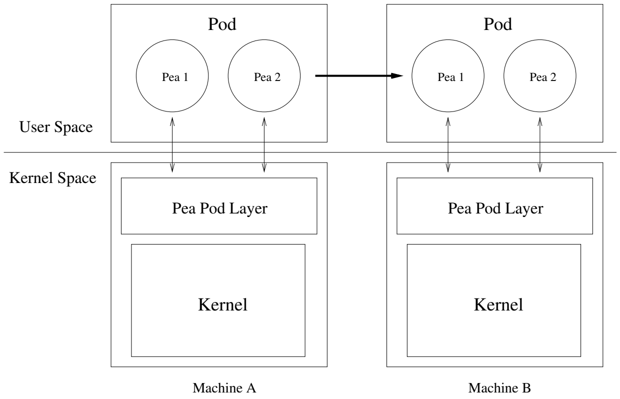

Figure 1: Pea-Pod migration

A pod namespace is virtual in that all operating system resources including processes, user information, files, and devices are accessed through virtual identifiers within a pod. These virtual identifiers are distinct from host-dependent resource identifiers used by the operating system. The pod virtual namespace provides a host-independent view of the system by using virtual identifiers that remain consistent throughout the life of a process in the pod, regardless of whether the pod moves from one system to another. Since the pod namespace is separate from the underlying operating system namespace, the pod namespace can preserve this naming consistency for its processes even if the underlying operating system namespace changes, as may be the case in migrating processes from one machine to another.

The pod private, virtual namespace enables secure isolation of applications by providing complete mediation to operating system resources. Pods can restrict what operating system resources are accessible within a pod by simply not providing identifiers to such resources within its namespace. A pod only needs to provide access to resources that are needed for running those processes within the pod. It does not need to provide access to all resources to support a complete operating system environment. For example, a pod can easily provide a least privilege environment tailored to the needs of an application services. If one had a web server that just served up static content, one could easily setup the pod to only contain the files the web server needs to run as well as the content it wants to serve. If the web server application gets compromised, the pod limits the ability of an attacker to further harm the system since the only resources he has access to are the ones explicitly needed by the service. Since the pod namespace provides the same application interface as the underlying operating system, pods can provide complete mediation without modifying, recompiling, or relinking applications.

The pod private, virtual namespace enables process migration by providing a consistent, host-independent view of the underlying operating system. Operating system resource identifiers such as process IDs (PIDs) must remain constant throughout the life of a process to ensure its correct operation. However, when a process is moved from one operating system to another, there is no guarantee that the underlying operating system will provide the same identifiers to a migrated process; those identifiers may in fact already be used by other processes in the system. The pod namespace addresses these issues by providing consistent, virtual resource names in place of host-dependent resource names such as PIDs. Names within a pod are trivially assigned in a unique manner in the same way that traditional operating systems assign names, but such names are localized to the pod. Since the namespace is private to a given pod, there are no resource naming conflicts for processes in different pods. There is no need for the pod namespace to change when the pod is migrated, which allows pods to ensure that identifiers remain constant throughout the life of the process, as required by legacy applications that use such identifiers.

A process can run inside a pod, but there are times when it is desirable to further restrict a process inside a pod in terms of the pod resources it can access. For example, in a conventional e-mail system, one will have a privileged SMTP daemon, such as Sendmail, and a non-privileged delivery agent, such as Procmail. While the Sendmail server runs with privilege, it actually needs a very small resource namespace. However, the Procmail delivery agent can make use of programs, such as SpamAssassin, to enable users to filter their e-mail effectively. Since these two programs need to interact directly, they can not be run in separate pods. Peas are introduced for the purpose of allowing these programs to interact, while restricting them to smaller resource namespaces. A pea is an abstraction that can contain a subset of processes within a pod and restrict those processes to accessing only a subset of pod resources. Pods can contain a group of processes, but the group may be composed of interacting components with different resource needs. Peas can separate these components within the pod by providing fine-grained and dynamic resources restrictions on differing sets of processes. The pea abstraction allows for processes running within a pod to have varying levels of isolation among them by running them in separate peas.

A pea achieves isolation levels by controlling what resources of a pod its processes are allowed to access and interact with. Peas provide a “see, but don’t touch” resource restriction model. For example, a process in a pea may be able to see file system resources and processes available to other peas, but can be restricted from accessing them. Unlike processes in separate pods, processes in separate peas in a single pod can “see each other” in that they share the same namespace and can be allowed to interact using traditional interprocess communication mechanisms. Processes can also be allowed to move from one pea to another in the same pod. However, by default processes in separate peas “can’t touch” any resource outside of it’s pea, be it a process pid or file system entry. Peas can support a wide range of resource restriction policies. By default, processes contained in a pea can only interact with other processes in the same pea. They have no access to other resources, such as file system and network resources or processes outside of the pea. This provides for a set of fail safe defaults, as any extra access has to be explicitly allowed by the administrator.

Many peas can be running side by side to provide flexibility in implementing a least privilege system for programs that are composed of multiple components that must work together, but do not all need the same level of privilege. One usage scenario would be to have a severely resource limited pea in which a privileged process executes but allowing the process to use traditional Unix semantics to work with less privileged programs that are in less resource restricted peas. One use of this is the mail delivery services already described, one can create two separate peas for Sendmail and Procmail to run within. It can similarly be used to allow a web server the ability to serve dynamic content via CGI in a more secure manner. Since the web server and the CGI scripts need separate levels of privilege, as well as different resource requirements, they shouldn’t have to run within the same security context. By configuring two separate peas for a web service, one for the web server to run within, and a separate for the specific CGI programs it wants to execute, one limits the damage that can occur if a fault is discovered within the web server. If one manages to execute malicious code within the context of the web server, one can only make use of resources that are allocated to the web server’s pea, as well as only execute the specific programs that are needed as CGIs. Since the CGI programs will also only run within their specific security context, the ability for malicious code to do harm is severely limited.

Peas and pods together provide secure isolation based on flexible resource restriction for programs as opposed to restricting access based on users. Pea-Pods also do not subvert underlying system restrictions based on user permissions, but instead complement such models by offering additional resource control based on the environment in which a program is executed. Instead of allowing programs with root privileges to do anything they want to a system, Pea-Pods enable a system to control the execution of such programs to limit their ability to harm a system even if they are compromised. Pea-Pods provide program-based resource restriction for file access, device access, network access, root privileges, process interactions, process transitions among peas, and resource utilization. Pea-Pods can restrict root privileges by disallowing certain operating system services for a given pea or pod. Pea-Pods can restrict process interactions by disallowing interprocess communication with processes outside of a pod, and by limiting such interactions among processes in separate peas in a pod. Pea-Pods can dynamically control the ability of processes to transition between peas, enabling processes to have different dynamic privileges during their execution. Pea-Pods can control the resources that processes consume in a pea or pod to limit denial of service attacks against the system. Due to space constraints, the Pea-Pod resource usage model is not discussed further in this paper.

3 Pea-Pod Virtualization

To support the Pea-Pod abstraction design of secure and isolated namespaces on commodity operating systems, we employ a virtualization architecture that operates between applications and the operating system, without requiring any changes to applications or the operating system kernel. This virtualization layer is used to translate between the Pea-Pod namespaces and the underlying host operating system namespace. It also protects the host operating system from dangerous privileged operations that might be performed by processes within the Pea-Pod, as well as protecting those processes from processes outside of the Pea-Pod on the host. Pea-Pod virtualization is used to provide isolation of peas and pods as well as enable pods to be migratable. The virtualization support for pod migration is based on Zap [28].

3.1 Pod Virtualization

Pods are supported using virtualization mechanisms that translate between pod virtual resource identifiers and operating system resource identifiers. Every resource that a process in a pod accesses is through a *virtual name* which corresponds to an operating system resource identified by a *physical name*. When an operating system resource is created for a process in a pod, such as with process or IPC key creation, instead of returning the corresponding physical name to the process, the pod virtualization layer catches the physical name value, and returns a private virtual name to the process. Similarly, any time a process passes a virtual name to the operating system, the virtualization layer catches it and replaces it with the appropriate physical name. The key pod virtualization mechanisms used are a system call interposition mechanism and the `chroot` utility with file system stacking for file system resources.

Pod virtualization employs system call interposition to wrap existing system calls to check and replace arguments that take virtual names with the corresponding physical names before calling the underlying original system call. Similarly, the wrapper is used to capture physical name identifiers that the original system calls return and return corresponding virtual names to the calling process running inside the pod. Pod virtual names are maintained consistently as a pod migrates from one machine to another and are remapped appropriately to underlying physical names that may change as a result of migration. Pod system call interposition also masks out processes inside of a pod from processes outside of a pod to remove any interprocess host dependencies across pod boundaries. System call interposition is used to virtualize operating system resources including process identifiers, keys and identifiers for IPC mechanisms such as semaphores, shared memory, and message queues, and network addresses.

Pod virtualization uses system call interposition to determine the network accessibility of pod processes. Pods provide the same semantic interface to applications as regular machines, which provide Internet accessible and localhost addresses. Therefore, pods also provide two types of networking addresses. Pods provide one that is only accessible to processes in a pod and one that is accessible on the Internet. A pod restricts its processes to the set of network addresses given to the pod by using the same virtual to physical mapping concepts of PID and IPCs. Processes within a pod make use of a virtual name for a network address. Since the regular pod virtualization rules take affect, processes are confined to the appropriate addresses.

Pod virtualization employs the `chroot` utility and file systems stacking to provide each pod with its own file system namespace that can be separate from the regular host file system. The pod file system can be composed from loopback mounts from the host for pods that are only checkpointed and restarted on the same machine. Similarly, one can make use of a portable hard drive that one moves between the different hosts one wants to migrate within. More commonly, the pod file system is composed from remote mounts via a network file system such as NFS so that the same files can be made consistently available as a pod is migrated from one machine to another. More specifically, when a pod is created or moved to a host, a private directory named according to a pod identifier is created on the host to serve as a staging area for the pod's virtual file system. Within this directory, the various network-accessible directories that the pod is configured to access will be mounted from a network file server. For example, from a Unix-centric viewpoint, this set of directories could include `/etc`, `/lib`, `/bin`, `/usr`, and `/tmp`. The `chroot` system call is then used to set the staging area as the root directory for the pod, thereby achieving file system virtualization with negligible performance overhead. This method of file system virtualization provides an easy way to restrict access to files and devices from within a pod. This can be done by simply not including file hierarchies and devices within the pod file system namespace. If files and devices are not mounted within the pod virtual file system, they are not accessible to pod processes.

Because commodity operating systems are not built to support multiple namespaces, a security issue that pod virtualization must address is that there are many ways to break out of a standard chrooted environment, especially if one allows the `chroot` system call to be used by processes in a pod. Pod file system virtualization enforces the chrooted environment and ensures that the pod's file system is only accessible to processes within the given pod by using a simple form of file system stacking to implement a barrier. File systems provide a permission function that determines if a process can access a file. For example, if a process tries to access a file a few directories below the current directory, the permission function is called on each directory as well as the file itself in order. If any of calls determine that the process doesn't have permission on a directory, the chain of calls end. Even, if the permission function determines that the process would have access to the file itself, it must have permission to walk the directory hierarchy to the file to access it. We implement a barrier by simply stacking a small pod-aware file system on top of the staging directory that overloads the underlying permission function to prevent processes running within the pod from accessing the parent directory of the staging directory, and to prevent processes running only on the host from accessing the staging directory. This effectively confines a process in a pod to the pod's file system by preventing it from ever walking past the pod's file system root.

While any network file system can be used with pods to support migration, we focus on NFS because it is the most commonly used network file system. Pods can take advantage of the user identifier (UID) security model in NFS to support multiple security domains on the same system running on the same operating system kernel. For example, since each pod can have its own private file system, each pod can have its own `/etc/passwd` file that determines its list of users and their corresponding UIDs. In NFS, the UID of a process determines what permissions it has in accessing a file. By default, pod virtualization keeps process UIDs consistent across migration and keeps process UIDs the same in the pod and operating system namespaces. However, since the pod file system is separate from the host file system, a process running in the pod is effectively running in a separate security domain from another process with the same UID that is running directly on the host system. Although both processes have the same UID, each process is only allowed to access files in its own file system namespace. Similarly, multiple pods can have processes running on the same system with the same UID, but each pod effectively provides a separate security domain since the pod file systems are separate from one another.

The pod UID model supports an easy-to-use migration model when a user may be working in one administrative domain and then moves to another. Even if the user has computer accounts in both administrative domains, it is unlikely that the user will have the same UID in both domains if they are administratively separate. Nevertheless, pods can enable the user to run the same pod with access to the same files in both domains. Suppose the user has UID 100 on a machine in administrative domain A and starts a pod connecting to a file server residing in domain A. Suppose that all pod processes are then running with UID 100. When the user moves to a machine in administrative domain B where he has UID 200, he can migrate his pod to the new machine and continue running processes in the pod. Those processes can continue to run as UID 100 and continue to access the same set of files on the pod file server, even though the user's real UID has changed. While this example considers the case of having a pod with all processes running with the same UID, it is easy to see that the pod model supports pods that may have running processes with many different UIDs.

Because the root UID 0 is privileged and treated specially by the operating system kernel, pod virtualization also treat UID 0 processes inside of a pod in a special way to prevent them from breaking the pod abstraction, accessing resources outside of the pod, and causing harm to the host system. While a pod can be configured for administrative reasons to allow full privileged access to the underlying system, we focus on the case of pods for running application services which do not need to be used in this manner. Pods do not disallow UID 0 processes, which would limit the range of application services that could be run inside pods. Instead, pods provide restrictions on such processes to ensure that they function correctly inside of pods.

While a process is running in user space, the UID it runs as doesn't have any effect. Its UID only matters when it tries to access the underlying kernel via one of the kernel entry points, namely devices and system calls. Since a pod already provides a virtual file system that includes a virtual `/dev` with a limited set of secure devices, the device entry point is already secured. The only system calls of concern are those that could allow a root process to break the pod abstraction. Only a small number of system calls can be used for this purpose. Pod virtualization classifies these system calls into three classes that need to be protected.

The first class of system calls are those that only affect the host system and serve no purpose within a pod. Examples of these system calls include those that load and unload kernel modules or that reboot the host system. Since these system calls only affect the host, they would break the pod security abstraction by allowing processes within it to make system administrative changes to the host. System calls that are part of this class are therefore made inaccessible by default to processes running within a pod.

The second class of system calls are those that are forced to run unprivileged. Just like NFS, by default, squashes root on a client machine to act as user `nobody`, pod virtualization forces privileged processes to act as the `nobody` user when it wants to make use of some system calls. Examples of these system calls include those that set resource limits and `ioctl` system calls. Since system calls such as `setrlimit` and `nice` can allow a privileged process to increase its resource limits beyond predefined limits imposed on pod processes, privileged processes are by default treated as unprivileged when executing these system calls within a pod. Similarly, the ioctl system call is a system call multiplexer that allows any driver on the host to effectively install its own set of system calls. Since the ability to audit the large set of possible system calls is impossible given that pods may be deployed on a wide range of machine configurations that are not controlled by the PeaPod system, pod virtualization conservatively treats access to this system call as unprivileged by default.

The final class of system calls are calls that are required for regular applications to run, but have options that will give the processes access to underlying host resources, breaking the pod abstraction. Since these system calls are required by applications, the pod checks all their options to ensure that they are limited to resources that the pod has access to, making sure they aren't used in a manner that breaks the pod abstraction. For example, the mknod system call can be used by privileged processes to make named pipes or files in certain application services. It is therefore desirable to make it available for use within a pod. However, it can also be used to create device nodes that provide access to the underlying host resources. To limit how the system call is used, the pod system call interposition mechanism checks the options of the system call and only allows it to continue if it's not trying to create a device.

3.2 Pea Virtualization

Peas are supported using virtualization mechanisms that impose levels of isolation among processes running within a single pod in separate peas by labeling resources and enforcing a simple set of configurable rules. For example, when a process is created in a pea, its process identifier is tagged with the identifier of the pea in which it was created. A process's ability to access pod resources is then dictated by the set of rules associated with its pea. Like pod virtualization, the key pea virtualization mechanisms used are a system call interposition mechanism and file system stacking for file system resources.

Pea virtualization employs system call interposition to wrap existing system calls to enforce restrictions on process interactions by controlling access to process and IPC virtual identifiers. Since each resource is labeled with the pea in which it was created, the system call interposition mechanism simply checks if the pea labels of the calling process and the resource to be touched are the same or different, providing an effective means of enforcing the pea's "see, but don't touch" model. For example, if a process in one pea would try to send a signal to another process in a seperate pea by using the kill system call, the system would return an error value of EPERM, as the process exists, just this process has no permission to signal it. On the other hand, a parent is able to use the wait system call to wait on a child process, even if that child process is running within a seperate pea since wait doesn't "touch" a process by affecting its execution.

When a new program is executed one might want to switch pea security domains. Peas support a single type of pea specific rule that let a pea determines how a process can transition from one its own pea to another. This rule is specified by a program filename and pea identifier. A pea may have multiple rules of this type. The rule specifies that a process should be moved into the pea specified by the pea identifier if it executes the program specified by the given filename. This is useful when it is known what a process will execute and it is desirable to have that program execution occur in an execution environment with different resource restrictions. For example, an Apache web server running in a pea may want to execute its CGI child processes in a more restrictive pea. This is supported via system call interposition by intercepting the exec system call and changing peas if a matching pea transition rule is specified for the pea in which the calling process is executing. Note that pea transition rules are one-way transitions that do not enable a process to return to its previous pea unless its current pea explicitly provides such rules.

System call interposition is also used to control network access for processes inside the pea. Peas provide two networking rules, one to allow processes in the pea to make outgoing network connections on a pod's virtual network adapters, the other to allow processes in the pea to bind to specific ports on the adapter to receive incoming connections. Pea rules can allow complete access to a pod network adapter, or only allow access on a per port basis. Since any network access occurs through system calls, peas simply check the options of the networking system call to ensure that it is allowed to perform the specified action.

Pea virtualization employs a set of file system rules and file systems stacking to provide each pea with its own permission set on top of the pod file system. To provide a least privilege environment, processes shouldn't have access to file system privileges they don't need. For example, while Sendmail has to write to /var/spool/mqueue, it only has to read its configuration from /etc/mail and should not need to have write permissions on its configuration. To implement such a least privilege environment, peas enable files to be tagged with additional permission rules that overlay the respective underlying file permissions. File system permissions determine access rights based on the user identity of the process while pea file permission rules determine access rights based on the pea context in which a process is executed. Each pea file rule can selectively allow or deny use of the underlying read, write and execute permissions of a file on a per pea basis. The underlying file permission is always enforced, but pea permissions can further restrict whether the underlying permission is allowed to take effect. The final permission is achieved by performing a bitwise AND operation on both the pea and file system permissions. For example, if the pea permission allowed for read and execute, the permission set of r-x would be triplicated to r-xr-xr-x- for the 3 sets of Unix permissions and the bitwise AND operation would effectively mask out any write permission that the underlying file system might allow. This prevents any process in the pea from opening the file and modifying it.

Enforcing on disk labeling of every single file is intractable if the underlying file system is going to be used for multiple disparate pods and peas. Since each pea in each pod might make use of similar underlying files but have different permission schemes, storing the pea permission data on disk effectively is not feasible. Instead, peas support the ability to dynamically label each file within a pod's file system based on two simple path matching rules, *path specific rules* and *directory default rules*. A path specific rule matches an exact path on the file system. For instance, if there's a path specific rule for `/home/user/file`, only that file will be matched with the appropriate permission set. On the other hand, if there's a directory default rule for the directory `/home/user/` any file under that directory in the directory tree can match it, and inherit its permission set.

Given a set of path specific and directory default rules for a pea, the algorithm for determining what rule matches to what path starts with the complete path and walks up the path to the root directory until it finds a matching rule. The algorithm can be described in four simple steps:

1. If the specific path has a *path specific rule*, return that rule set.

2. Otherwise, choose the path's directory as the current directory to test.

3. If the directory being tested has a *directory default rule*, return that rule set.

4. Otherwise set its parent as the current directory to test and go back to step 3.

This ensures that if there's no *path specific rule*, the closest *directory default rule* to the specified path becomes the rule for that path. Also, since by default peas give the root directory "/" a *directory default rule* denying all permissions, the default for every file on the system, unless otherwise specified is deny, ensuring a fail safe default setup.

The semantics of pea file permission rules are based on file path name. If a file has more than one path name, such as via a hard link, both have to be protected by the same rule, otherwise depending on how the underlying file is accessed the permission set it gets will be non-deterministic as the inode cache will contain the permission set of the path name that was opened initially. This is only an issue on setup of a Pea-Pod, as once its setup, any hard links that are created will obey the regular file system rules, which include being unable to hard link to a path one's pea doesn't have access to, as well as any new hard link path name that gets created is given a path specific rule equivalent to the original path's rule.

The pea architecture makes use of the pod's stackable file system to integrate the pea file system namespace restrictions into the regular kernel permission model. It accomplishes this by stacking on top of the file system's *lookup* function which fills in the respective file's inode structure, and the *permission* function which makes use of the stored permission data to make simple permission determinations. Since a file system's permission function is a standard part of the operating system kernel's security infrastructure, no changes have to be made to the kernel's file system security infrastructure.

The stackable file system uses a unique set of hash tables that it organizes in a tree structure to mimic the underlying file system. Every directory can be represented by a hash table, and entries in the hash table correspond to directory entries that have pea file system rules. If a directory entry is an actual directory, it would have a corresponding child hash table. Looking up the appropriate rule for any path name is simply parsing the path name into directory entry tokens, and performing a token by token traversal of the tree of hash tables. This traversal results in finding the rule that best matches the pathname, based on the decision algorithm given above. Since hashing of tokens is fast, one can quickly traverse the tree in $O(h)$ time, where h is the height of the file system tree, no matter how many rules the file system enforces. The stackable file system is made even faster by the fact that the rule lookup doesn't have to be done often, since we store the data in the file system's inode structure and the kernel caches the inode structure for later use.

4 Migration Across Different Kernels

To maintain application service availability without losing important computational state as a result of system downtime due to operating system upgrades, Pea-Pods provide a checkpoint-restart mechanism that allows pods to be migrated across machines running different operating system kernels. Upon completion of the upgrade process, the respective Pea-Pod and its applications are restored on the original machine now with an upgraded operating system. We assume here that the systems have not been compromised and that any kernel security holes on the unpatched system have not yet been exploited on the system; migrating across kernels that have already been compromised is beyond the scope of this paper.

We also limit our focus to migrating between machines with a common CPU architecture with kernel differences that are limited to maintenance and security patches. These patches often correspond to changes in the minor version number of the kernel. For example, the Linux 2.4 kernel has more than twenty minor versions. Even within minor version changes, there can be significant changes in kernel code. Table 1 shows the number of files that have been changed in various subsystems of the Linux 2.4 kernel

Type	.c Files	Changed	Percentage
Drivers	2221	2079	93.6
Arch	2694	2351	87.2
FS	524	488	93.1
Network	422	352	83.4
Core Kernel	27	22	81.4
VM	20	20	100
IPC	4	4	100

Table 1: Kernel Changes within the Linux 2.4 Series

across different minor versions. For example, all of the files for the VM subsystem were changed since extensive modifications were made to implement a completely new page replacement mechanism in Linux. Many of the Linux kernel patches contain security vulnerability fixes, which are typically not separated out from other maintenance patches. We similarly limit our focus to where the application's execution semantics, such as how threads are implemented and how dynamic linking is done, do not change. On the Linux kernels this is not an issue as all these semantics are enforced by user-space libraries. Whether one uses kernel or user threads, or one how libraries are dynamically linked into a process is all determined by the respective libraries on the file system. Since the Pod has access to the same file system on whatever machine it is running on, these semantics stay the same.

To support migration across different kernels, Pea-Pods use a checkpoint-restart mechanism that employs an intermediate format to represent the state that needs to be saved on checkpoint. On checkpoint, the intermediate format representation is saved and digitally signed to enable the restart process to verify the integrity of the image. Although the internal state that the kernel maintains on behalf of processes can be different across different kernels, the high-level properties of the process are much less likely to change. We capture the state of a process in terms of higher-level semantic information specified in the intermediate format rather than kernel specific data in native format to keep the format portable across different kernels. For example, the state associated with a Unix socket connection consists of the directory entry of the Unix socket file, its superblock information, a hash key, and so on. It may be possible to save all of this state in this form and successfully restore on a different machine running the same kernel. But this representation of a Unix socket connection state is of limited portability across different kernels. A different high-level representation consisting of a four tuple, virtual source pid, source fd, virtual destination pid, destination fd is highly portable. This is because the semantics of a process identifier and a file descriptor is typically standard across different kernels, especially across minor version differences.

The intermediate representation format used by Pea-Pods for migration is chosen such that it offers the degree of portability needed for migrating between different kernel minor versions. If the representation of state is too high-level, the checkpoint-restart mechanism could become complicated and impose additional overhead. For example, the Pea-Pod system saves the address space of a process in terms of discrete memory regions called VM areas. As an alternative, it may be possible to save the contents of a process's address space and denote the characteristics of various portions of it in more abstract terms. However, this would call for an unnecessarily complicated interpretation scheme and make the implementation inefficient. The VM area abstraction is standard across major Linux kernel revisions. Pea-Pods view the VM area abstraction as offering sufficient portability in part because the organization of a process's address space in this manner has been standard across all Linux kernels and has never been changed since its inception.

Pea-Pods further support migration across different kernels by leveraging higher-level native kernel services to transform intermediate representation of the checkpointed image into an internal representation suitable for the target kernel. Continuing with the previous example, Pea-Pods restore a Unix socket connection using high-level kernel functions as follows. First, two new processes are created with virtual PIDs as specified in the four tuple. Then, each one creates a Unix socket with the specified file descriptor and one socket is made to connect to the other. This procedure effectively recreates the original Unix socket connection without depending on many kernel internal details.

This use of high-level functions helps in general portability of using Pea-Pods for migration. Security patches and minor version kernel revisions commonly involve modifying the internal details of the kernel while high-level primitives remain unchanged. As such services are usually made available to kernel modules through exported kernel symbol interface, the Pea-Pod system is able to perform cross-kernel migration without requiring modifications to the kernel code.

The Pea-Pod checkpoint-restart mechanism is also structured in such a way to perform its operations when processes are in a state that checkpointing can avoid depending on many low-level kernel details. For example, semaphores typically have two kinds of state associated with each of them: the value of the semaphore and the wait queue of processes waiting to acquire the corresponding semaphore lock. In general, both of these pieces of information have to be saved and restored to accurately reconstruct the semaphore state. Semaphore values can be easily obtained and restored through GETALL and SETALL parameters of the `semctl` system call. But saving and restoring the wait queues involves manipulating kernel internals directly. The Pea-Pod mechanism avoids having to save the wait queue information by requiring that all the processes be stopped before taking the checkpoint. When a process waiting on a semaphore receives a stop signal, the kernel immediately releases the process from the wait queue and returns

EINTR. This ensures that the semaphore wait queues are always empty at the time of checkpoint so that they do not have to be saved.

While Pea-Pods can abstract and manipulate most process state in higher-level terms using higher-level kernel services, there are some parts that not amenable to a portable intermediate representation. For instance, specific TCP connection state like timestamp values and sequence numbers, which do not have a high-level semantic value, have to be saved and restored in order to maintain a TCP connection. As this internal representation can change, its state needs to be tracked across kernel versions and security patches. Fortunately, there is usually an easy way to interpret such changes across different kernels because networking standards such as TCP do not change often. Across all of the Linux 2.4 kernels, there was only one change in TCP state that required even a small modification in the Pea-Pod migration mechanism. Specifically, in the Linux 2.4.18 kernel, an extra field was added to TCP connection state to address a flaw in the existing syncookie mechanism. If configured into the kernel, syncookies protect an Internet server against a synflood attack. When migrating from an earlier kernel to Linux-2.4.18, the Pea-Pod system initializes the extra field in such a way that the integrity of the connection is maintained. In fact, this was the only instance across all of the Linux 2.4 kernel versions where an intermediate representation was not possible and the internal state had changed and had to be accounted for.

To provide proper support for Pea-Pod virtualization when migrating across different kernels, we must ensure that that any changes in the system call interfaces are properly accounted for. As pea-pods have a virtualization layer using system call interposition mechanism for maintaining namespace consistency and ensuring pea security, a change in the semantics for any system call intercepted by pea-pods could be an issue in migrating across different kernel versions. But such changes usually do not occur as it would require that the libraries be rewritten. In other words, Pea-Pod virtualization is protected from such changes in a similar way as legacy applications are protected. However, new system calls could be added from time to time. Such system calls could have implications to the pea encapsulation mechanism. For instance, across all Linux 2.4 kernels, there were two new system calls, `gettid` and `tkill` for querying the thread identifier and for sending a signal to a particularly thread in a thread group, respectively, which needed to be accounted for to properly virtualize Pea-Pods across kernel versions. As these system calls take identifier arguments, they were simply intercepted and virtualized.

5 Security Analysis and Examples

Saltzer and Schroeder[37] describe several principles for designing and building secure systems. These include:

- *Economy of mechanism*: Simpler and smaller systems are easier to understand and ensure that they do not allow unwanted access.
- *Fail safe defaults*: Systems must choose when to allow access as opposed to choosing when to deny.
- *Complete mediation*: Systems should check every access to protected objects.
- *Least privilege*: A process should only have access to the privileges and resources it needs to do its job.
- *Psychological acceptability*: If users are not willing to accept the requirements that the security system imposes, such as very complex passwords that the users are forced to write down, security is impaired. Similarly, if using the system is too complicated, users will misconfigure it and end up leaving it wide open.
- *Work factor*: Security designs should force an attacker to have to do extra work to break the system. The classic quantifiable example is when one adds a single bit to an encryption key, one doubles the key space an attacker has to search.

Pea-Pods are designed to satisfy these six principles. Pea-Pods provide economy of mechanism using a thin virtualization layer based on system call interposition and file system stacking that only adds a modest amount of code to a running system. The largest part of the system is due to the use of a null stackable file system with 7000 lines of C code, but this file system was generated using a simple high-level file system language [45], and only 50 lines of code were added to this well tested file system to implement the Pea-Pod file system security. Furthermore, Pea-Pods change neither applications nor the underlying operating system kernel. The modest amount of code to implement Pea-Pods makes the system easier to understand. Since the Pea-Pod security model only provides resources that are explicitly stated, it is relatively easy to understand the security properties of resource access provided by the model.

Furthermore, Pea-Pods provide fail safe defaults by only providing access to resources that have been explicitly given to peas and pods. Since Pea-Pod virtualization limits access to the underlying system to its virtual namespace, Pea-Pods provide complete mediation to operating system resources. Peas in pods are explicitly designed to provide least privilege by restricting programs in an environment that can be easily limited to provide the least amount of access for the encapsulated program to do its job. Pea-Pods provide psychologically acceptability by providing users and system administrators with a standard system environment where all they have to understand are their applications and the system resources that they need without detailed understanding of any underlying operating system specifics.

Similar to least privilege, Pea-Pods increase the work factor that it would take to compromise a system by simply not making available the resources that attackers depend on to harm a system once they have broken in. For example, since Pea-Pods can provide selective access to what program are included within their view, it would be very difficult to get a root shell on a system that does not have access to any shell program. Similarly, the fact that one can migrate a system away from a host that is vulnerable to attack increases the work an attacker would have to do to make services unavailable.

We briefly describe three examples that help illustrate how Pea-Pods can be used to improve computer security and application availability for different application scenarios. The application scenarios are e-mail delivery, web content delivery, and desktop computing.

For e-mail delivery, Pea-Pods can isolate different components of e-mail delivery to provide a significantly higher level of security in light of the many attacks on Sendmail vulnerabilities that have occurred. Consider isolating a Sendmail installation that also provides mail delivery and filtering via Procmail. E-mail delivery services are often run on the same system as other Internet services to improve resource utilization and simplify system administration through server consolidation. However, this can provide additional resources to services that do not really need them, potentially increasing the damage that can be done to the system if attacked. Using Pea-Pods, both Sendmail and Procmail can execute in the same pod, which isolates e-mail delivery from other services on the system. Since pod's allow one to migrate a service between machines, the e-mail delivery pod is migratable. If a fault is discovered in the underlying host machine, the e-mail delivery service can be moved to another system while the original host is patched, preserving the availability of the e-mail service.

Furthermore, Sendmail and Procmail can be placed in separate peas which facilitate necessary interprocess communication mechanisms between them while improving isolation. This pod is a common example of a privileged service that has child helper applications. In this case, the Sendmail pea is configured with full network access to receive e-mail, but without shell access since there is no reason why Sendmail needs a shell. Sendmail would be denied write access to file system areas such as `/usr/bin` to prevent modification to those executables, and would only be allowed to transition a process to the Procmail pea if it is executing Procmail. On mail delivery, Sendmail would then `exec` Procmail in the Procmail pea, which would be configured with more liberal access to process shell scripts and run other programs such as SpamAssassin. As a result, the Sendmail/Procmail pod can provide full e-mail delivery service while isolating Sendmail such that even if Sendmail is compromised by an attack, such as a buffer overflow, the attacker would be contained in the Sendmail pea and not even be able to execute a root shell to attempt to further compromise the system.

Note that there are multiple ways to configure Internet services peas. With the e-mail delivery example, we illustrated a simple system configuration to prevent the common buffer overflow exploit of getting the privileged server to execute a local shell. By simply denying access to shells but allowing access to other files, we limit the amateur attacker's ability to exploit flaws, while requiring very little configuration or knowledge of the actual services. On the other hand, one can also use Pea-Pods to create a complete least privilege environment to contain more professional attackers to the domain they exploited.

For web content delivery, Pea-Pods can isolate different components of web content delivery to provide a significantly higher level of security in light of common web server attacks that may exploit CGI script vulnerabilities. Consider isolating an Apache web server front end, a MySQL database backend, and CGI scripts that interface between them. While one could run Apache and MySQL in seperate pods, since they are providing a single service, it make sense to run them within a single pod that can be migrated as a unit. If the underlying host comes under attack, such as via a denial of service attack, one can use the pod's migration mechanism to move the web content delivery pod to a safer machine, providing better service availability in a hostile environment. However, since both Apache and MySQL are within the pod's single namespace, if an exploit is discovered in Apache, it could be used to perform unauthorized modifications to the MySQL database.

To provide greater isolation among different web content delivery components, we can use three peas in a pod: one for Apache, a second for MySQL, and a third for the CGI programs. Each pea is configured to contain the minimal set of resources needed by the processes running within the respective pea. The Apache pea includes the apache binary, configuration files and the static html content, as well as a rule to exec all CGI programs into the CGI pea. The CGI pea contains the relevant CGI programs as well as access to the MySQL daemon's named socket, allowing interprocess communication with the MySQL daemon to perform the relevant SQL queries. The MySQL pea contains the mysql daemon binary, configuration files and the files that make up the relevant databases. Since Apache is the only program exposed to the outside world, it is the only process that can be directly exploited. However, if an attacker is able to exploit it, the attacker is limited to a pea that is only able to read or write specific Apache files, as well as exec specific CGI programs into a seperate pea. Since the only way to access the database is through the CGI programs, the only access to the database an attacker would have is what is allowed by said programs. Consequently, it becomes very difficult to cause serious harm to such a Pea-Pod web content delivery system.

For desktop computing, Pea-Pods enable desktop computing environments to accommodate mobile users across separate administrative domains. As users move from one geographic location to another, Pea-Pods allow them to take their computing with them in a hassle-free way. Since Pea-Pods provide complete mediation as well as fail safe defaults, system administrators can allow desktop computing pods from separate security domains to migrate onto their hosts, since the processes within the pod are prevented from harming it and can be configured to only access files from the pod file system securely exported to remote machines via NFS over IPSec. Peas can also be used within the context of such a desktop computing environment to provide additional isolation. Many application used on a daily basis, such as mp3 players and web browsers, have had security holes in the past that could possibly enable attackers to cause them to execute malicious code or give them access to the entire local file system [19, 20].

To secure an mp3 player, an mp3 player pea can be created within a desktop computing pod that restricts the mp3 player's ability to make use of files outside of a special mp3 directory. Since most users store their music within its own subtree, this isn't a serious restriction. Most mp3 content should not trusted, especially if one is streaming mp3s from a remote site. By running the mp3 player within this fully restricted pea, a malicious mp3 cannot compromise the user's desktop session. This mp3 player pea is simply configured with three file system rules. A path specific rule that provides access to the mp3 player itself is required to load the application. A directory default rule that provides access to the entire mp3 directory subtree is required to give the process access to the mp3 file library. Finally, a path specific rule that provides access to the `/dev/dsp` audio device is required to allow the process to actually play audio.

To secure a web browser, a web browser pea can be created within a desktop computing pod that restricts the web browser's access to system resources. Consider the Mozilla web browser as an example. A Mozilla pea would need to have all the files Mozilla needs to run accessible from within the pea. Moziall dynamically loads libraries itself and stores them along with its plugins within the `/usr/lib/mozilla` directory. By providing a directory default rule that provides access to that directory, as well as another directory default rule that provides access to the user's `.mozilla` directory, the Mozilla web browser can run as normal within this special Mozilla pea. One would also want the ability to be able to download and save files, as well as launch viewers, such as for postscript or mp3 files, directly from the web browser. This involves a simple reconfiguration of Mozilla to change its internal `application.tmp_dir` variable to be a directory that is within the Mozilla pea. By creating such a directory, such as `downloads` within the users home directory, and providing a directory default rule allowing access, we enable one to explicitly save files, as well as as implicitly save when one wants to execute a helper application. Similarly, just like Mozilla is configured to run helper applications for certain file types, one would have to configure the Mozilla pea to execute those helper applications within their respective peas. As shown for an mp3 player, configuring such a pea for these process is fairly simple. The only addition one would have to make is to provide an additional pea transition rule to the Mozilla pea that tells the Pea-Pod system to transition the process to a separate pea on execution of programs such as the `mpg123` mp3 player or the `gv` postscript viewer.

Name	**Description**	**Linux**
getpid	average `getpid` runtime	350 ns
ioctl	average runtime for the FIONREAD ioctl	427ns
shmget-shmctl	IPC Shared memory segment holding an integer is created and removed	3361 ns
semget-semctl	IPC Semaphore variable is created and removed	1370 ns
fork-exit	process forks and waits for child which calls exit immediately	44.7 us
fork-sh	process forks and waits for child to run `/bin/sh` to run a program that prints "hello world" then exits	3.89 ms
Apache	Runs Apache under load and measures average request time	1.2 ms
Make	Linux Kernel compile with up to 10 process active at one time	224.5s
Postmark	Use Postmark Benchmark to simulate Sendmail performance	.002s
MySQL	"TPC-W like" interactions benchmark	8.33s

Table 2: Application Benchmarks

6 Experimental Results

We implemented Pea-Pods as a loadable kernel module in Linux that requires no changes to the Linux kernel. We present some experimental results using our Linux prototype to quantify the overhead of using Pea-Pods on various applications. Experiments were conducted on a trio of IBM Netfinity 4500R machines, each with a 933Mhz Intel Pentium-III CPU, 512MB RAM, 9.1 GB SCSI HD and a 100 Mbps Ethernet connected to a 3Com Superstack II 3900 switch. One of the machines was used as an NFS server from which directories were mounted to construct the virtual file system for the Pea-Pods on the other client systems. The clients ran different Linux distributions and kernels, one machine running Debian Stable with a Linux 2.4.5 kernel and the other running Debian Unstable with a Linux 2.4.18 kernel.

To measure the cost of Pea-Pod virtualization, we used a range of micro benchmarks and real application workloads and measured their performance on our Linux Pea-Pod prototype and a vanilla Linux system. Table 2 shows the seven micro-benchmarks and four application bench-

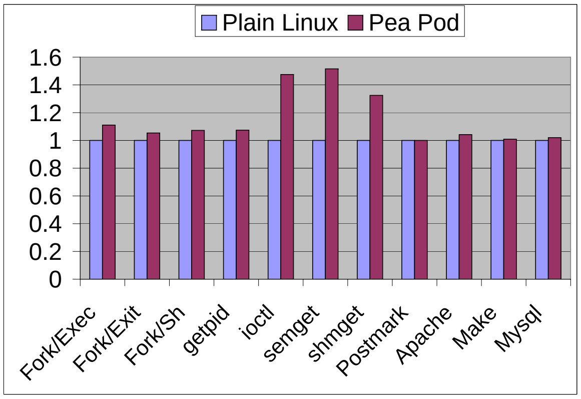

Figure 2: Pea-Pod Virtualization Overhead

Name	Applications
E-mail	Sendmail 8.12.3 with the pod configured to automatically change peas on execution of Procmail.
Web	Apache 1.3.26 and MySQL 3.23.49 running within separate peas inside the same Pod.
KDE	Xvnc – VNC 3.3.3r2 X Server KDE – Entire KDE 2.2.2 environment, including window manager, panel and assorted background daemon and utilities SSH – openssh 3.4p1 client inside a KDE konsole terminal connected to a remote host Shell – The Bash 2.05a shell running in a konsole terminal KGhostView – A PDF viewer with a 450k 16 page PDF file loaded. Konqueror – A modern standards compliant web browser that is part of KDE KOffice – The KDE word processor and spreadsheet programs

Table 3: Application Scenarios for Migration

Case	Checkpoint	Restart	Size	Compressed
E-mail	0.079s	0.049s	848KB	124KB
Web	0.308s	0.508s	5.3MB	332KB
KDE	0.851s	0.942s	35MB	8.8MB

Table 4: Pea-Pod Migration Costs

marks we used to quantify Pea-Pod virtualization overhead as well as the results for a vanilla Linux system. To obtain accurate measurements, we rebooted the system between measurements. Additionally, the system call micro-benchmarks directly used the TSC register available Pentium CPUs to record timestamps at the significant measurement events. Each timestamp's average cost was 58 ns. The files for the benchmarks were stored on the NFS Server. All of these benchmarks were performed in a chrooted environment on the NFS client machine running Debian Unstable with a Linux 2.4.18 kernel. Figure 2 shows the results of running the benchmarks under both configurations, with the vanilla Linux configuration normalized to one. Since all benchmarks measure the time to run the benchmark, a small number is better for all benchmarks results.

The results in Figure 2 show that Pea-Pod virtualization overhead is small. Pea-Pods incur less than 10% overhead for most of the micro-benchmarks and less than 4% overhead for the application workloads. The overhead for the simple system call `getpid` benchmark is only 7% compared to vanilla Linux, reflecting the fact that Pea-Pod virtualization for these kinds of system calls only requires an extra procedure call and a hash table lookup. The most expensive benchmarks for Pea-Pods is `semget+semctl` which took 51% longer than vanilla Linux. The cost reflects the fact that our untuned Pea-Pod prototype needs to allocate memory and do a number of namespace translations. The `ioctl` benchmark also has high overhead, because of the 12 separate assignments it does to protect the call against malicious root processes. This is large compared to the simple FIONREAD `ioctl` that just performs a simple dereference. However, since the `ioctl` is simple, we see that it only adds 200 ns of overhead over any `ioctl`. For real applications, the most overhead was only four percent which was for the Apache workload, where we used the `http_load` benchmark [30] to place a parallel fetch load on the server with 30 clients fetching at the same time. Similarly, we tested MySQL as part of a web-commerce scenario outlined by TPC-W with a bookstore servlet running on top of Tomcat with a MySQL back-end. The Pea-Pod overhead for this scenario was less than 2% versus vanilla Linux.

To measure the cost of Pea-Pod migration and demonstrate the ability of Pea-Pods to migrate real applications, we migrated the three application scenarios discussed in Section 5, an email delivery service using Sendmail/Procmail, a web content delivery service using Apache/MySQL, and a KDE desktop computing environment with an isolated web browser. Table 3 described the configurations of the application scenarios we migrated. To demonstrate our Pea-Pod prototype's ability to migrate across Linux kernels with different minor versions, we checkpointed each application workload on the 2.4.5 kernel client machine and restart it on the 2.4.18 kernel machine. For these experiments, the workloads were checkpointed to and restarted from local disk.

Table 4 shows the time it took to checkpoint and restart each application workload. In addition to these, migration time also has to take into account network transfer time. As this is dependent on the transport medium, we include the uncompressed and compressed checkpoint image sizes. In all cases, checkpoint and restart times were fast, taking less than a second for both operations, even when performed on separate machines or across a reboot. We also show that the actual checkpoint images that were saved were modest in size for complex workloads. For example, the KDE pod had over 30 different processes running, providing the desktop applications applications, as well as substantial underlying window system infrastructure, including inter-application sharing, a rich desktop interface managed by a window manager with a number of applications running in a panel such as the clock. Even with all these applications running, they checkpoint to a very reasonable 35 MB uncompressed for a full desktop environment. Additionally, if one needed to transfer the checkpoint images over a slow link, Table 4 how they can be compressed very well with the bzip2 compression program.

7 Related Work

Historically, the military has been concerned with confidentiality and controlling the flow of information. Bell and LaPadula [8] as well as Biba [9] formulated models that formalize the concepts of ensuring confidentiality and integrity constraints between programs running at different classification levels. The work was incorporated into Multics' Multilevel Security Model [22] and the later Orange Book specification [14]. This work on information flow [24] is orthogonal to Pea-Pods, which focuses on containing untrusted applications.

Language-based tools have been used to try to harden the applications against buffer overflow attacks. Examples of this include the StackGuard compiler [13] and the LibSafe [6] interposition library. Similarly, others have strived to encourage the use of safer languages and language features, such as the type safety of ADA and Java. While LibSafe can work with unmodified dynamically linked applications, the majority of these solutions require applications to be rewritten or recompiled. Pea-Pods compliment these approaches by providing isolation of legacy applications without modification.

Privilege separation [32, 4] is a programming model that can be used to help prevent malicious code from executing in a privileged context. By separating each task of a system into a small process, one can create multiple simple programs that work together to perform a complex task and are easier to verify for correctness. Since the system is split into multiple processes, each process can be given a restricted set of privileges based on what it needs to do. OpenSSH and Qmail are two program examples that implement privilege separation. The Pea-Pod sandbox provides a form of privilege separation for legacy processes without requiring a redesign of the application service.

NSA's Security Enhanced Linux [26], which is based upon the Flask Architecture [40], implements a policy language that one can use to implement models that enable one to enforce privilege separation. The policy language is very flexible, but this also makes them very complex. Their example security policy is over 80 pages long. There is research into creating tools to make policy analysis tractable [2], but the fact that the language is so complex makes it difficult for the average end user to construct an appropriate policy. Peas, like NSA SE Linux, operate on a resource level where every resource is tagged, while Pod's operate like a virtual machine where resources not allocated to the namespace are unavailable. Pods offer simplicity, such that even a novice administrator can determine what's available to both well behaved and malicious code. Peas provide the ability to provide simple increases in security, while also scaling up in complexity as required.

Janus [43, 17] and Systrace [31] are rule-based systems used for determining access controls. They implement system call interposition to control at an individual system call level what kernel functionality a process can use. Systrace provides graphical tools that help build rules on the fly. However, policy creation for Janus and Systrace requires a fine understanding of system calls. This provides great flexibility, but it makes them hard to configure, as well making final configurations difficult to understand. Like Pea-Pods, Janus and Systrace operate at the system call level. Unlike Pea-Pods, Janus and Systrace are also configured at the same individual system call level. Neither system integrates support for secure isolation with migration capabilities.

FreeBSD's Jail mode [21] implements a simpler to understand sandbox. It provides a chroot like environment that processes can not break out of. However, since Jail is limited in what it can do, such as the fact it doesn't allow IPC within a jail[16] many real world application will not work. Pea-Pods, on the other hand, do not place any restrictions on the types of applications that can run in its sandboxed environment.

SubDomain [12] creates a sandboxed view of the underlying file system for applications to run in. Like the pea-aware file system, it attempts to allow a system administrator to limit a processes' file system view to the minimum set needed by that application. However, since SubDomain's sandbox doesn't encapsulate processes, processes running as root can take advantage of system calls such as `signal` to affect change on processes outside their sandbox. While the Pea-Pods file system model is similar to SubDomain, it is conceptually different. While SubDomain operates at the system call level, the pea file system is a full-fledged file system. For example, when a file is opened, SubDomain must resolve it if it is a symbolic link. Pea-Pods, on the other hand, just uses the permission associated with the file at the end of the link as a regular file system does. Similarly, since Pea-Pods includes a full fledged file system, it integrates fully with the regular kernel security infrastructure and provides much better performance.

Virtual machine monitors (VMMs) can also be used to provide a secure sandbox environment [42, 44, 7]. VMMs can also be used to migrate an entire operating system environment [38]. Pea-Pods can compliment the functionality of VMMs. Unlike Pea-Pods, VMMs decouple processes from the underlying machine hardware, but tie them to an instance of an operating system. As a result, VMMs cannot migrate processes apart from that operating system instance and cannot continue running those processes if the operating system instance ever goes down, such as during security upgrades. In contrast, Pea-Pods decouple process execution from the underlying operating system which allows it to migrate processes to another system when an operating system instance is upgraded. Similarly, VMMs just provide a single operating system namespace and lack the ability to isolate components within an operating system. If a single process in a VMM is exploitable, malicious code can make use of it to access and make use of the entire set of operating system resources. Since Pea-Pod's decouple processes from the underlying operating system and it's resulting namespace, they are natively able to limit the separate processes of a larger system to the appropriate resources needed by them.

Many systems have been proposed to support process migration, but not in the context of supporting application availability in the presence of operating system patches and upgrades. Several such research operating systems [34, 27, 3, 36, 15, 5, 11] rely on a single system image across all machines for process migration, in addition to the ability to forward many operations to the home node. They do not provide migration across independent commodity operating systems. Several user-space migration systems have been designed to run on commodity operating systems [25, 33, 29, 10]. These systems are primarily designed for long running scientific computations and cannot support processes that use many standard operating system services, such as IPC. TUI [39] provides support for process migration across machines running different operating systems and hardware architectures. Unlike Pea-Pods, TUI has to compile applications on each platform using a special compiler and does not work with unmodified legacy applications. Pea-Pods build on Zap [28], which supports transparent migration across systems running the same kernel version. Unlike Zap, Pea-Pods provide pod security and support for isolating processes inside of a pod. Furthermore, Pea-Pods support transparent migration across different minor kernel versions, which is essential for providing application availability in the presence of operating system security upgrades.

Pea-Pods can be used to improve the security of trusted computing systems[41, 18], which can enable the operating system and third parties to determine the identity of a program and if it's authorized to be executed. However, if a fault is discovered within a running trusted program, an attacker can make use of that fault to inject untrusted code into the system enabling access to the full set of resources. For example, Microsoft's X-Box, which runs a trusted operating system on trusted hardware, enforces a policy of only loading authorized games. However, buffer overflows in the code of trusted games have enabled users to load an untrusted Linux kernel and use the X-Box as a normal computer [1]. Pea-Pods can be used to limit the resources available to faulty trusted programs and thereby further limit an attacker's ability to compromise a trusted computing system.

8 Conclusions

The Pea-Pod system provides an operating system virtualization layer that decouples process execution from the underlying operating system. The virtualization layer supports two key abstractions for encapsulating processes, peas and pods. Pods provide lightweight sandboxes that mirror the underlying operating system environment, and peas provide fine-grain least privilege environments within pods. Together, peas and pods can isolate untrusted applications within sandboxes, preventing them from being used to attack the underlying host system or other applications even if they are compromised. The Pea-Pod sandboxes can be transparently migrated across machines running different operating system kernel versions. This enables security patches to be applied to operating systems in a timely manner with minimal impact on the availability of sandboxed application services. Pea-Pod secure isolation and migration functionality is achieved without any changes to applications or operating system kernels. We have implemented Pea-Pods in a Linux prototype and demonstrated how peas and pods can be used to improve computer security and application availability for a range of applications, including e-mail delivery, web servers and databases, and desktop computing. Our results show that Pea-Pods can provide easily configurable, secure migratable sandboxes that can run a wide range of desktop and server Linux applications in least privilege environments with low overhead.

References

[1] Anonymous. Technical Analysis of 007: Agent Under Fire save game hack. `http://www.xbox-linux.org/docs/007analysis.html`, Jul 2003.

[2] M. Archer, E. Leonard, and M. Pradella. Towards a Methodology and Tool for the Analysis of Security-Enhanced Linux. Technical Report NRL/MR/5540—02-8629, NRL, 2002.

[3] Y. Artsy, Y. Chang, and R. Finkel. Interprocess communication in charlotte. *IEEE Software*, pages 22–28, Jan 1987.

[4] N. Associates. Privman - a library to make privilege separation easy. `http://opensource.nailabs.com/privman/`.

[5] A. Barak and R. Wheeler. MOSIX: An Integrated Multiprocessor UNIX. In *Proceedings of the USENIX Winter 1989 Technical Conference*, pages 101–112, San Diego, CA, Feb. 1989.

[6] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2000.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.

[8] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report ESD-TR-74-244, Mitre Corp, Bedford, MA, May 1973.

[9] K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, Mitre Corp, Bedford, MA, 1977.

[10] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A migration transparent version of PVM. *Computing Systems*, 8(2):171–216, 1995.

[11] D. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, Mar 1988.

[12] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *14th USENIX Systems Administration Conference (LISA 2000)*, New Orleans, LA, Dec. 2000.

[13] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan. 1998.

[14] Department of Defense. Trusted Computer System Evaluation Criteria (Orange Book). Technical Report DoD 5200.28-STD, Department of Defense, Dec. 1985.

[15] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementatio. *Software - Practice and Experience*, 21(8):757–785, Aug. 1991.

[16] FreeBSD Project. Developer's handbook. `http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/se%cure-chroot.html`.

[17] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proc. Network and Distributed Systems Security Symposium*, Feb. 2003.

[18] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, October 2003.

[19] GOBBLES Security. Local/remote mpg123 exploit. `http://www.opennet.ru/base/exploits/1042565884_668.txt.html`.

[20] GreyMagic Security Research. Reading local files in netscape 6 and mozilla. `http://sec.greymagic.com/adv/gm001-ns/`.

[21] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *2nd International SANE Conference*, MECC, Maastricht, The Netherlands, May 2000.

[22] P. A. Karger and R. R. Schell. MULTICS Security Evaluation: Vulnerability Analysis. Technical Report ESD-TR-74-193, Mitre Corp, Bedford, MA, June 1977.

[23] B. LaMacchia. Personal Communication, Jan 2004.

[24] C. E. Landwehr. Formal Models for Computer Security. *ACM Computing Surveys*, 13(3):247–278, Sept. 1981.

[25] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin Madison Computer Sciences, Apr. 1997.

[26] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.

[27] S. J. Mullender, G. v. Rossum, A. S. Tanenbaum, R. v. Renesse, and H. v. Staveren. Amoeba a distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[28] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dec. 2002.

[29] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. In *Proceedings of Usenix Winter 1995 Technical Conference*, pages 213–223, New Orleans, LA, Jan 1995.

[30] J. Poskanzer. `http://www.acme.com/software/http_load/`.

[31] N. Provos. Improving Host Security with System Call Policies. In *12th USENIX Security Symposium*, Washington, DC, Aug. 2003.

[32] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, Washington, DC, August 2003.

[33] J. Pruyne and M. Livny. Managing checkpoints for parallel programs. In *2nd Workshop on Job Scheduling Strategies for Parallel Processing (In Conjunction with IPPS '96)*, Honolulu, Hawaii, Apr. 1996.

[34] R. Rashid and G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th Symposium on Operating System Principles*, pages 64–75, Dec 1984.

[35] E. Rescorla. Security holes... Who cares? In *Proceedings of the 12th USENIX Security Conference*, Washington, D.C., Aug. 2003.

[36] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA), 1992.

[37] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Fourth ACM Symposium on Operating System Principles*, Oct. 1973.

[38] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[39] P. Smith and N. C. Hutchinson. Heterogeneous process migration: The Tui system. *Software – Practice and Experience*, 28(6):611–639, 1998.

[40] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proc. of the Eighth USENIX Security Symposium*, Aug. 1999.

[41] Trusted Computing Platform Alliance. TCPA main specification v1.1b. `http://www.trustedcomputing.org`.

[42] VMware, Inc. `http://www.vmware.com`.

[43] D. Wagner. Janus: an approach for confinement of untrusted applications. Master's thesis, University of California, Berkeley, 1999.

[44] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dec. 2002.

[45] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.