Ostia: A Delegating Architecture for Secure System Call Interposition

Tal Garfinkel Ben Pfaff Mendel Rosenblum
{talg,blp,mendel}@cs.stanford.edu
Computer Science Department, Stanford University

Abstract

Application sandboxes provide restricted execution environments that limit an application's access to sensitive OS resources. These systems are an increasingly popular method for limiting the impact of a compromise. While a variety of mechanisms for building these systems have been proposed, the most thoroughly implemented and studied are based on system call interposition. Current interposition-based architectures offer a wide variety of properties that make them an attractive approach for building sandboxing systems. Unfortunately, these architectures also possess several critical properties that make their implementation error prone and limit their functionality.

We present a study of Ostia, a sandboxing system we have developed that relies on a "delegating" architecture which overcomes many of the limitations of today's sandboxing systems. We compare this delegating architecture to the "filtering" architecture commonly used for sandboxes today. We present the salient features of each architecture and examine the design choices that significantly impact security, compatibility, flexibility, deployability, and performance in this class of system.

1 Introduction

Today's applications, from email clients to web servers, are often vulnerable to attack. Buffer overflows, back doors, logic errors, or simple misconfigurations permit attackers to compromise these applications with disturbing frequency. A heavily studied approach to remedying this problem is running programs in application sandboxes [18, 2, 3, 31, 30, 12, 10, 15, 28], i.e. execution environments that impose application-specific restrictions on access to system resources.

Hybrid interposition architectures, so called because they rely on both a kernel-level enforcement mechanism and a user-level policy engine, have been a prominent approach to building these tools. These systems leverage the hardware memory protection that operating systems offer to provide a high assurance and efficient mechanism for isolating the address spaces of sandboxed applications from the rest of the system. A relatively simple mechanism can allow a user-level sandbox program to interpose on the system call interface of a sandboxed application, allowing regulation of access to all sensitive system resources including the file system and network.

Filtering sandboxing architectures have been the dominant approach to building hybrid sandboxes. In these systems a user-level sandboxing program confines other applications by interposing on their access to the system call interface via a kernel-level process tracing mechanism [2, 3, 18, 29, 31, 21, 30, 1]. The sandboxing program can then filter the flow of system calls between the application and the OS. Unfortunately, these tools have suffered from a variety of security problems which has limited their functionality and made their design and implementation particularly error prone [16]. In this work we demonstrate that these shortcomings are not a fundamental property of hybrid sandboxes, but rather an artifact of several properties of filtering-based architectures.

We present an alternative architecture that we call a *delegating architecture* that retains the benefits of a filtering approach, while overcoming many of its limitations. With a delegating architecture, instead of a sandboxed application requesting sensitive resources directly from the kernel, it delegates responsibility for obtaining sensitive resources to the program ("agent") controlling the sandbox. This agent accesses resources on behalf of the sandboxed program according to a user-specified security policy.

To motivate the need for a delegating architecture, as well as to highlight the salient features of this class of system, we compare Ostia, our implementation of a sandbox with a delegating architecture, to J2 (Janus version 2), a sandbox we previously developed based on a filtering architecture. Through this comparison we show how delegating architectures can greatly simplify the task of system call interposition and provide greater flexibility and assurance than current approaches.

The next section provides a deeper discussion of hybrid sandboxes, delving further into the properties that motivate our interest in these systems. Section 3 presents the significant features of this class of system and describes both filtering and delegating sandboxes, highlighting the salient features of each. In section 4 we provide a detailed description of J2 and Ostia and explore their implementations. Section 5 evaluates and compares each architecture's impact on security, policy flexibility, compatibility with existing software, ease of deployment, and performance. We discuss related work in section 6 and give our conclusions in section 7.

2 Motivation

A system call interposition-based sandbox can be constructed using a spectrum of mechanisms for isolation, interposition, and policy.

Purely user-level sandboxes can be realized through software-based isolation techniques, such as software-based fault isolation (SFI) [14], program shepherding [23], software dynamic translation (SDT) [32], and safe languages. Because these systems do policy enforcement at user level, excellent extensibility can be realized without any need to modify the OS kernel. However, these approaches manifest a number of limitations. Safe languages (e.g. Java) and low-level software-based techniques (e.g. SFI) are often extremely specific to a particular API or ABI, greatly limiting the range of languages and architectures they can support. Further, the greater complexity of software-based isolation mechanisms provides less assurance than simpler hardware-based mechanisms. Finally, these mechanisms often impose a non-trivial overhead on program execution.

Strictly OS-based mechanisms that reside entirely in the kernel [24, 28, 12, 5] and rely on hardware memory protection for isolation can also be used. OS-based isolation is fast, offers excellent assurance, and is already provided by standard operating systems. Further, OS-based isolation does not depend on the internal APIs or ABIs of the software, and can be used in conjunction with software-based techniques (e.g. safe languages). However, placing an entire sandboxing system in the kernel also has undesirable consequences. A sandboxing system can still be a nontrivial addition to the kernel and past errors in such systems have introduced new security vulnerabilities [34]. The size and complexity of internal interfaces in modern monolithic kernels and their rate of change make it more difficult to gain confidence in the correctness of purely kernel-based solutions. The irregular and dynamic nature of kernel internals also greatly exacerbate problems of portability, auditing, and code maintenance. Finally, user-level code offers a much richer development environment (e.g. languages, libraries, debuggers) which greatly simplifies development [17]. All these factors contribute to making a purely kernel-level solution undesirable from assurance, extensibility, and maintenance standpoints.

In a hybrid sandbox [31, 18, 2], kernel-level code provides support for leveraging OS isolation and provides basic enforcement mechanisms, while the remaining portion of the system resides at user level. A hybrid approach enjoys many of the attractive properties of both kernel- and user-level approaches. Leveraging hardware memory protection provided by the OS kernel yields greater assurance, better compatibility, etc. compared to purely user-level solutions, while keeping most of the sandbox at user level provides safe extensibility, eases development and maintenance, etc. Unfortunately, current hybrid systems have a number of critical shortcomings that make their design and implementation complex and error prone, and limit their flexibility. We demonstrate that these shortcomings are not an essential property of hybrid sandboxes but rather an artifact of today's filtering architectures. We demonstrate that in delegating architectures these shortcomings can be mitigated, and the desirable properties of a hybrid approach enhanced.

The most significant problem that filtering-based sandboxes exhibit is that they are prone to race conditions [16]; this is a fundamental property of their architecture. In these systems, permission checking performed by the sandbox is separate from access granting by the OS. As a result, state (e.g. call arguments, file system metadata, etc.) used by a system call can be changed between the time of the check by the sandbox and its use by the OS. Modifying this state can allow an attacker to change the object that a call refers to, leading to a "time of check/time of use" (TOCTOU) race condition that attackers can exploit [33, 7].

Filtering sandboxes have attempted to alleviate the problem of races by pushing more sandbox functionality into the kernel and by not supporting particularly problematic classes of programs (e.g. multi-threaded applications). This can allow filtering-based systems to alleviate some of these races on a piecemeal basis, but this increasingly mitigates the benefits a hybrid approach. Further, as a filtering sandbox desires more control over the semantics of system calls in order to express richer policies (e.g. rewriting system call arguments), further support must be pushed into the kernel, again on a feature-by-feature basis. Clearly, there is some fundamental conflict arising between the requirements of the sandboxing system, and the functionality provided at user level by a filtering approach.

A delegating sandboxing architecture resolves this conflict by providing more power to the user level sandboxing system. Instead of simply providing an interface to filter system calls (i.e. to allow or deny calls like a packet filter), a delegating architecture completely virtualizes those portions of the system call interface that the sandbox interposes on. This provides the user level sandbox complete control over how resources are accessed, as the sandbox actually performs access to the resources on the sandboxed programs behalf. This approach alleviates the significant security problems of filtering sandboxes. For example, because the sandboxing system has complete control over accessing resources it can ensure that this takes place in a manner that excludes the possibility of race conditions. Section 5.1 provides a complete discussion of how delegation helps alleviate races and facilitates a more conservative and secure design in a variety of other ways. Delegation also greatly enhances extensibility as it allows system calls to be arbitrarily redefined/transformed, without ever requiring kernel changes e.g. rewriting system call arguments could trivially be facilitated at user level (for further details see section 5.2). Finally, because a delegating architecture moves virtually all of its functionality to user level, it requires only trivial kernel support. For example, our implementation required only 200 lines of code. The requirement for minimal kernel support enhances the security, portability, and maintainability properties of a hybrid sandbox.

3 Hybrid interposition architectures

At a high level, hybrid interposition-based sandboxes have two components: the interposition architecture and the policy engine. The policy engine is responsible for interpreting a user-specified policy and deciding which resources the sandboxed application should be allowed to access. The interposition architecture is responsible for providing the functionality required by the policy engine to make decisions (e.g. provide access to system call arguments) and enforce those decisions. Policy engines can be made largely independent of the interposition architecture.

Policy decisions are made by interpreting the meaning of a system call (i.e. what resource it will grant access to) then making a decision based on a user-specified sandbox policy. System calls are regulated based on the policy model, which specifies which calls to allow or deny, given the sandbox policy. The sandbox policy is typically a file consisting of a set of rules specifying which resources an application may access. Appendix A describes the the format for these rules (or policy interface) used by Ostia, which is the same as that used by J2 for ease of comparison.

More complicated examples of policy interfaces can be found in other systems [31, 2]. We have intentionally kept the policy interface of J2 and Ostia simple as we believe that a more baroque policy format would merely serve as a distraction from our focus on system architecture.

Policy model: Hybrid interposition-based sandboxes leverage the isolation provided by the OS's process abstraction. All security-sensitive interactions between the sandboxed application and the system outside its address space are conducted via the system call interface. (Minor exceptions such as core dumps are easily accounted for.)

While the UNIX API is quite large, we only need to regulate the modest number of calls that have an impact outside of the process. The majority of this attention goes to regulating access to the network and file system. The remaining sensitive calls are easy to handle as they have few parameters and are generally allowed or denied outright, regardless of their arguments.

The UNIX model for providing access to the network and file system is largely based on a simple capability model where obtaining a capability, called a "descriptor," for a resource (e.g. file descriptor with the `open` call) is performed via an operation separate from resource use. Thus we are primarily concerned with controlling calls that acquire these descriptors (e.g. `open`, `socket`) or modify them (e.g. `bind`, `setsockopt`). We do not interpose on calls that simply use descriptors (e.g. `read`, `write`) or copy existing descriptors (e.g. `dup`). This separation is important for achieving good performance.

The initial process in a sandbox is started with an essentially empty descriptor space. Subsequent processes started in the sandbox must either obtain descriptors for resources by explicitly requesting them over a checked interface or by inheriting them from a parent, whose accesses were also checked. Thus we can explicitly control a sandboxed process's descriptor set. (Descriptors can also be obtained from other processes via the `sendmsg` and `recvmsg` calls. These calls are also regulated through policy.)

Some prominent examples of other calls which are checked include: calls that manipulate file system metadata (e.g. `rename`, `remove`), calls that modify sandboxed processes' user and group identities (e.g. `setuid`), and calls to send signals (e.g. `kill`).

Execution model: The lifetime of a program in an application sandbox progresses in similar steps, regardless of the architecture. To start a program in the sandbox, the user invokes the sandbox specifying the program to run and the policy to apply. The user-level portion of the system reads the policy and starts a child. The child releases its resources (file descriptors, etc.), performs some action to "enter" the sandbox, then `execs` the sandboxed application. The parent (called the "monitor" or "agent" depending on sandbox type) then enters an event-handling loop that receives requests to access resources in the form of system calls. It allows or denies these requests according to decisions made by the policy engine. The policy engine in turn makes decisions by interpreting the requests in the context of the current system state; how it obtains this state is architecture dependent. The sandboxed application consists of one or more processes (called "clients") which make requests for resources. These may be made to a single parent which multiplexes all requests or multiple parents. The lifetime of the sandbox ends when no client processes remain.

Concurrency strategy: The monitor(s) in a filtering sandbox or agent(s) in a delegating sandbox must be able to receive and answer requests sent concurrently by multiple processes, so the sandbox developer must make a decision about how to handle this concurrency. The two primary options are to multiplex them through a single process using `select` or a similar mechanism, or to handle them concurrently with multiple processes or threads. The choice of concurrency strategy can significantly impact complexity and scalability. We discuss this further in sections 4.2 and 5.5 respectively.

3.1 Filtering architectures

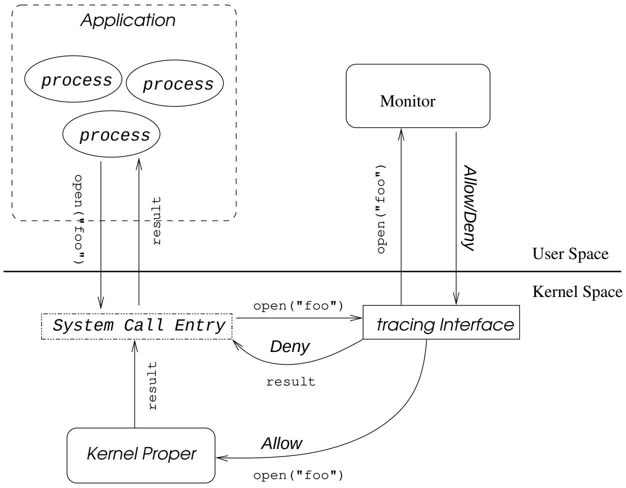

Figure 1: Filtering architecture

Most existing application sandboxes have a filtering architecture, illustrated by Figure 1. It consists of two parts: a kernel-based tracing mechanism to filter the system calls of a sandboxed application, and a user-level "monitor" that tells the tracing interface which calls to allow or deny based on a user-specified policy.

In a filtering sandbox, when a sandboxed process ("client") executes a sensitive call, the process tracing mechanism puts it to sleep and sends a request to the monitor. The monitor responds to the request with "allow" or "deny" based on the policy engine's judgment. The tracing mechanism then wakes up the sandboxed process. If the call is allowed, the client's call proceeds normally. If the call is denied, the call is forced to return an error code immediately. Calls which are not deemed sensitive by the monitor are never trapped by the tracing interface, and thus execute as they would normally in an unsandboxed application.

3.2 Delegating architectures

Our new sandbox, Ostia, has the delegating architecture depicted in Figure 2. It has two primary parts: a kernel portion that enforces a hard-coded policy preventing all calls that provide direct access to sensitive resources (e.g. `open`, `socket`) from being executed, and a user-level portion

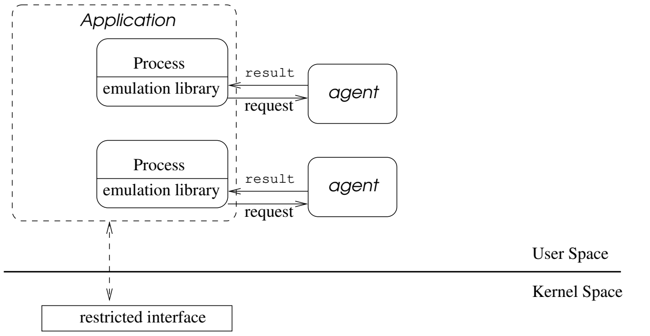

Figure 2: Delegating architecture

("agent") that performs access to sensitive resources on behalf of the sandboxed process ("client") where permitted by the policy engine. These systems usually have a third part, that we refer to as the emulation library. The emulation library resides in the address space of sandboxed processes. It converts a sandboxed process's sensitive system calls into IPC requests to the agent. How exactly this is done is implementation dependent (section 4.2 discusses the approach we use).

When a client makes a sensitive system call, it is redirected to the emulation library, which sends a request to its agent via an IPC channel. If the request is permitted by policy, the agent accesses the requested resource (possibly executing one or more system calls) and returns the result (e.g. return code, descriptor) to the client. As in a filtering sandbox, calls which do not provide access to sensitive resources but merely use resources the client has already obtained (e.g. `read`, `write`) are executed directly by the client.

The fact that the agent both checks permissions and accesses the requested resource on the child's behalf is the most important distinction between the agent in a delegating sandbox and the monitor in a filtering sandbox. The delegating sandbox gets its name from the fact that the ability to access sensitive resources is revoked from the client and delegated to the agent.

4 Implementations

This section describes in more detail the implementation of the relevant parts of Ostia, our delegating sandbox. To make our comparisons against filtering sandboxes more concrete, we also briefly describe J2, our filtering sandbox. More detailed descriptions of Janus [36, 18], J2 [16, 29], and other very similar filtering architectures [2, 3] are available elsewhere. We present further details in later sections as they become relevant.

4.1 J2

J2 (Janus version 2) is a canonical example of a filtering architecture. It was developed through successive rewritings of the original Janus system and retains its basic structure. It differs most prominently in its use of a dedicated process tracing mechanism, `mod_janus`, specifically designed for secure interposition, instead of relying on an existing process tracing interface.

Tracing mechanism: J2's tracing mechanism, `mod_janus`, was originally developed in response to the shortcomings of existing process tracing mechanisms for supporting secure system call interposition [36].

`mod_janus` provides a simple interface for the monitor process. To sandbox a process the monitor attaches a descriptor to it and specifies which system calls to trap and which to allow. The monitor calls `select` on the descriptors associated with its sandboxed processes to poll for trap events. Trap events are generated when the sandboxed process makes a "trapped" system call. When a trap event is pending on a descriptor, the type of call that was trapped and call arguments can be read from the descriptor. Once a process has generated a trap it is put into an uninterruptible sleep state and can only continue once given an "allow" or "deny" by the monitor. It is impossible for a process to escape the sandbox; closing a descriptor kills its process, and descriptors cannot be unbound.

Most trapped events are entries into system calls. An exception is `fork`, whose exit is trapped to allow the monitor to attach to the new child process. To ensure that the child cannot execute any calls outside the sandbox, `mod_janus` ensures that the monitor attaches to it before it is allowed to begin execution.

When a system call is trapped, call arguments (e.g. path names and `struct sockaddrs`) are immediately copied out of the process's address space and into a per-process kernel buffer. When pathnames are copied into the kernel they are resolved (canonicalized) with symlinks being expanded in the context of the trapped process. This ensures that canonicalization takes place in the proper namespace, i.e. if the process is `chroot`ed, makes a reference to `/proc/self`, or there is some other per-process variation in the name space, this will be taken into account. The kernel is redirected to this internal copy of the arguments for evaluating the call. Copying arguments into the kernel prevents arguments from being modified which could lead to certain types of race conditions. It does not prevent other kinds of races, as will be discussed in section 5.1.

Concurrency strategy: In a filtering sandbox like J2, both single-threaded `select`-based and multithreaded architectures are feasible. The original Janus prototype used one monitor process per sandboxed process. J2's monitor uses a multiplexing model to handle concurrency, in which a single monitor process polls for client requests with `select` followed by a `read` from the descriptor associated with the pending request. The decision was made to go to a `select`-based model in J2 as a result of the belief that this would substantially reduce overhead under load. However, as we discuss in section 5.5, this seems to have actually hurt scalability because the single monitor becomes a performance bottleneck.

4.2 Ostia

Ostia implements a delegating sandbox architecture. As described in the previous section, it is composed of three primary components.

Kernel module: A small kernel module enforces Ostia's static policy of denying any call that provides direct access to sensitive system resources. This is done simply by preventing a fixed set of system calls from executing. (As a belt-and-suspenders measure to ensure that access to the file system is denied, sandboxed processes are `chroot`ed to an empty directory if Ostia is run as root.)

It also provides a trampoline mechanism that redirects delegated calls back into the emulation library as discussed below. Finally, it implements an `fexecve` call because `execve` cannot be delegated to another process, for obvious reasons.

Emulation library: Ostia uses a callback mechanism in the kernel module to redirect system calls. (Ostia evolved from an earlier delegating system we built that relied on shared library replacement to redirect system calls. We note this to emphasize that system call redirection—or virtualization—can be done multiple ways.) When a sensitive system call reaches the kernel entry point, it calls back into the handler in a special emulation library in the program's address space. The emulation library transforms the system call into a request to the agent. To speed up subsequent system calls from the same point in the code, the handler also examines the machine instructions that made the call and, if they take the expected form, patches them in-place to jump directly to the handler, avoiding subsequent round trips through the kernel.

Ostia's handler must be installed into the program's address space before the program gets control. It must be available even before the loader for dynamic libraries takes control, so that access to dynamic libraries can go through the agent. Ostia does this by implementing its own ELF binary loader in user space. Instead of executing the sandboxed program directly, it executes the loader program, which contains the emulation library and a startup routine. The startup routine registers the handler, manually loads the sandboxed program with `mmap` calls, and turns over control. The emulation library ensures that this happens on every `execve` by a client.

A process's emulation library sends requests (similar to RPC calls) to its respective agent over a UNIX domain socket. UNIX domain sockets are more than simply an interface for passing messages. They also allow file descriptors to be passed between process and agent. This feature is critical as it permits delegation of obtaining capabilities (e.g. open files) to the agent, while permitting processes to operate on capabilities (e.g. reading and writing files) directly.

Agents: As discussed in section 3, agents are responsible for reading the policy file, starting the initial sandboxed process, making policy decisions, etc. Each sandboxed process has its own agent. The most important function that an agent provides to its sandboxed process (or "client") is handling requests for calls from the emulation library.

System calls can be divided into three classes: calls that must be delegated, calls that are always permitted, and calls that are completely disallowed. Refer back to the policy model given in section 3 for additional background on the reasoning behind each category. Each sandboxed process has an agent to handle its delegated calls. Delegated calls fall into a few subcategories:

- **File system and network operations:** In Unix, files and network sockets are often used (read, written, etc.) via descriptors. Applications are always started with a descriptor space containing only the standard input, output, and error descriptors. This ensures that applications can only gain access to resources explicitly permitted by the sandbox.

  Any operation that refers to resources by name (i.e. a file by path name or network host by address) and not by descriptor must be delegated.

  Calls that refer to resources by name and grant access to descriptors (e.g. `open`, `socket`) are delegated by requesting the descriptor from the agent. For example, when the agent receives an `open` request, it first checks policy. If the `open` is permitted, the agent opens the file and passes the descriptor to the sandboxed process.

  Calls that refer to resources by name but do not grant access to descriptors include `rename`, `chmod`, `mkdir` and `sendto`. These are delegated by executing the operation in the agent. In this case no descriptor is returned. However, as with all delegated calls, a return value is passed back to the client reflecting the result returned by the system call, e.g. an error such as `EPERM`. As an exception, `sendmsg` and `recvmsg` on a Unix domain socket between a client and its agent are allowed via direct system calls to permit communication with the agent.

  Calls that modify the properties of objects referred to by descriptors already held by a client (e.g. `ioctl`, `bind`) are delegated by passing the object's descriptor to the agent. The agent can query the descriptor for the object's state (via e.g. `getpeername` or `fstat`), and if the modification conforms to the agent's policy, modify the object and return a success code to the requesting process.

  Calls that operate on a descriptor's object, but do not change its security relevant properties (e.g. `read`, `write`, `fstat`) are not delegated. Similarly, calls that modify a process's descriptor space but do not grant access to new resources (e.g. `dup2`, `close`) are also not delegated. As discussed in section 3, doing so is unnecessary and could incur significant performance overhead.

  `execve` is an odd corner case, where a call refers to a file by name but cannot be delegated. We addressed this by adding an `fexecve` call via the kernel module.

  The agent must take care to ensure that the operations it performs involving file names are not subject to race conditions. We discuss this issue further in section 5.1. How this is achieved is OS dependent for some calls. For a general treatment of this issue, refer to Viega [35].

- **File system state tracking:** When an agent accesses a resource on a sandboxed process's behalf, it must adopt or emulate all relevant properties of the process. Key properties for delegating file system operations are the current working directory, file creation mask (umask), and effective identity (euid, egid, and extended group membership). The agent must emulate these properties of the sandboxed process to emulate normal file system interface semantics.

  For this reason, the agent handles `chdir`, `umask`, and `getcwd` system calls, among others. These operations are delegated simply by examining and updating data structures within the agent that track the sandboxed process's state.

  To ensure that file system requests are interpreted correctly, the agent assumes the relevant file system state of its client before interpreting or fulfilling a request.

- **Id management:** To correctly perform accesses on the process's behalf, we need to know its user and group identities. There is no reason to let a process manipulate this state in the kernel, as it is no longer able to access sensitive resources directly. Thus, we prefer to run it completely without privilege and instead manage this state in the agent. To fool the process into believing it is still running under the normal OS privilege model we delegate this interface, which includes `setuid`, `setgid`, and `getuid`, to the agent which emulates the OS model for modifying these permissions. As with file system state tracking, these operations are delegated simply by examining and updating agent data structures.

When the agent performs a call on the sandboxed process's behalf, it simply assumes the appropriate identity based on the emulated permissions. Thus, normal OS access controls are enforced by the kernel. In spite of concerns instilled by other work [9] that this might be particularly error prone, we did not find implementing this to be difficult or intricate. The code is relatively clean and simple, and largely taken directly from the Linux kernel.

- **Signals:** The sandboxed process cannot be permitted to send signals directly. Instead signals are sent by delegating the responsibility for the `kill` call to the agent, which only permits signals to be sent to other processes in the sandbox and otherwise maintains normal signal semantics.

  A client process can make system calls that do not access sensitive system resources in the normal fashion, e.g. queries for information not typically considered security sensitive, such as `getpid` and `gettimeofday`. Operations that modify process state in safe ways of no interest for delegation purposes, such as `signal` and `ulimit`, are also permitted to execute normally.

- **`fork` handling and thread support:** The `fork` system call requires special handling. When the client invokes `fork`, the emulation library takes control and notifies the agent. The agent `forks` a second agent process and replies to the client with a UNIX domain socket descriptor for communicating with the new agent. Then the client calls into the kernel to perform the real client `fork`. Afterward, each client closes one of the descriptors.

  As for thread support, with a filtering architecture, sandboxes must provide extra code to prevent shared state from leading to races as discussed in section 5.1. In contrast, delegating sandboxes must provide extra code to share state between agents where necessary. Ostia needs such extra code only for thread support.

  In particular, the current working directory and file creation mask can be shared between multiple threads in a single program. When one of these threads sends a `chdir` request to its agent, the change in current working directory must be reflected in all of the agents. The agents cannot themselves be threads that share a single current working directory. Use of `chdir` is an essential part of checking file system policy, and serializing those uses across the agents would induce a performance hit. Instead, each agent checks between processing requests whether another agent has changed the current working directory and if so updates its own.

  The emulation library also needs support for threads. Threads can share a file descriptor table, so a different file descriptor must be used to connect each thread in a process to its agent. Each thread needs a piece of thread-specific data that designates the file descriptor for its agent. We support this type of thread-local storage through `mod_ostia`.

Concurrency strategy: Whereas a filtering sandbox can easily be implemented using a multiplexing or multithreaded concurrency model, Ostia exhibits a multithreading model, i.e. one agent process per sandboxed process, from necessity. In a delegating sandbox the agent both checks policy and executes approved operations. Under the multiplexing model it would serialize both policy checking and operation execution, which can cause correct programs to fail. Consider a pair of producer-consumer client processes that communicate over sockets with `sendmsg` and `recvmsg`, operations that must be checked for policy and can block in the server. If the consumer process runs and blocks waiting for input from the producer, it will wait forever because the producer will never get a chance to run.

This limitation does not appear to be a liability. In our experience, a multithreading sandbox is simpler and cleaner than the multiplexing equivalent, because each agent or monitor only manages state for a single process. Others have reported the same observation [36]. Also, a multiplexing sandbox can impose significant performance restrictions under high load due to serializing all requests on a single thread. This is examined further in section 5.5.

5 Evaluation

In this section we evaluate and compare Ostia and J2, considering the implications of these results for filtering and delegating architectures in general.

5.1 Security

Race conditions: Time-of-check/time-of-use ("TOCTOU") races [7] are a significant potential problem for sandboxing systems. These races occur when a policy engine performs a check to authorize a system call that relies on an object that a name (e.g. a file system path) references, but the name changes to refer to a different object before the operating system executes the call. This can occur when the name is stored as some type of shared state, e.g. when the policy engine checks that a given file name refers to an allowed file, but the file name changes to refer to a symbolic link before the operating system executes it. Races arise from three kinds of state:

1. **Inter-thread shared state**: State shared between multiple threads within a process, e.g. entire process memory space, user and group identity, current working directory, and file descriptor space.
2. **Inter-process shared state**: State shared between multiple threads or processes, e.g. memory shared with System V shared memory and `mmap` mechanisms.
3. **Globally shared state**: State shared by all processes on the system, e.g. the file system.

The key property of all of these forms of shared state is that any of them can change asynchronously from the perspective of a given thread. Put another way, regardless of whether a given thread is scheduled, these aspects of its state can change.

Race conditions in filtering sandboxes: Race conditions are a significant problem in sandboxes based on a filtering architecture, and no system, including J2, has fully addressed this problem. An in-depth study of this problem has been presented elsewhere [16]. We will review the main issues here to provide adequate context and appreciate the importance of this problem in filtering sandboxes.

When threads in a process share a single file descriptor table, the object a descriptor number references can change between check and use. Similarly, if two threads share a current working directory, then a thread's current working directory can be changed by a second thread between check and use. There does not appear to be any simple way to fix these races in filtering sandboxes. J2 simply disallows execution of multithreaded programs.

Shared memory (inter-thread and inter-process) results in argument races, i.e. races where an argument could change after it is checked by the policy engine, but before it is used by the system call. This is a problem for non-scalar system call arguments such as `struct sockaddrs` and pathnames, that typically reside in the sandboxed process's memory until they are used by the system call. As described in section 3, J2's solution, the same as that adopted by many other filtering sandboxes, is to marshal non-scalar arguments into protected kernel memory. This provides an adequate solution to the argument race problem, but it comes at a the cost of simplicity. The code to perform this functionality accounts for about 25% of `mod_janus`'s code.

Globally shared state in the file system is also a troublesome source of races. These race conditions come in two types: symbolic link races and relative path races.

Symbolic link races occur because any component in a path may be replaced by a symbolic link between time of check and time of use. Currently we are not aware of any implemented solution to this problem in a filtering sandbox. All published proposed solutions rely on canonicalizing the path name before it is checked, either in user space or in the kernel. This does not solve the problem; any component of the path can still change to a symbolic link, no matter how many times canonicalization is done.

Relative path races, the second type of file system race, can occur when the parent directory of a process's current working directory changes and a relative path is in use. Canonicalizing file names before use does solve this kind of race, as this forces the use of an absolute path. J2 performs this action and is thus immune to relative path races.

Clearly, solving some types of race conditions, possibly all on a piecemeal basis, is possible in a filtering sandbox. However, it comes at a great cost to implementation complexity, primarily in the kernel where it is least desirable. The complexity of these races and their solutions casts significant doubt on the security of these systems. It was many years after the first filtering-based sandboxing paper [18] that all of the aforementioned races were brought to light [16]. We may still be overlooking others.

Ameliorating races with delegation: Delegation alone does not prevent all races. However, it does prevent some, by placing inter-process/inter-thread state under control of the agent by default. It also easily facilitates the prevention of remaining races by giving the agent control over how resources are accessed. Let us consider how delegation allows each class of race to be easily addressed in Ostia:

- **Inter-thread and inter-process shared state races**: In a delegating sandbox, sensitive system calls are performed by the agent, so the file descriptor space, current working directory, etc., used by sensitive system calls are held exclusively by the agent. Most races, such as argument races, are no longer a concern because an external process cannot modify this state. One concern is whether an agent could be tricked into inducing a race because of state shared between multiple agent processes. As we noted in section 4.2, this is not a significant issue because agents only share the current working directory and file creation mask between multiple threads, explicitly and in a race-free fashion.
- **Globally shared state**: In some sense the primary problem that filtering sandboxes face is that they are not in control of how programs gain access to resources. Programs should be able to access resources in a race-free fashion, but the responsibility for ensuring race-free accesses falls upon the application programmer.

  If all programs carefully avoided file races, then a filtering sandbox would not need to worry about race conditions. For example, if all `open` calls in Linux were done with the "no follow" flag (which prevents symlink expansion in the last component of a path) then a filtering sandbox would not have to worry about the last component of a path being a symlink, one precondition for a race free open. Of course, not all programs make their calls following this convention.

In a delegating sandbox we can address this problem because the sandbox makes all accesses to resources itself. Thus accesses can be performed in a manner respecting OS conventions for providing race-free operations on the file system. Another way to view this is that the agent is an active proxy which normalizes calls to the OS to put them into a form which will provide a predictable result.

If the delegating sandbox naively opened files it would be prone to race conditions, just like poorly written programs in today's systems that suffer from the normal user-level file system races such as `/tmp` races [7]. By respecting OS conventions for safe file access, Ostia is able to obtain the descriptor to a known file, in particular one permitted by policy, while being safe from race conditions.

Code complexity: There is no simple way to summarize the security of a system. A popular starting point for comparison is lines of code. The counts given below are total lines of code (LOC) as determined by Brian Marick's `lc` program [26], rounded to the nearest 100. Code is written in C except where otherwise specified.

First, consider user-level code. The Ostia agent consists of 3,200 lines total. Of this, 700 comprise the policy engine, and the remaining 2,500 lines are the system core. The J2 monitor is effectively 3,000 LOC (1,400 LOC in the policy engine and 1,600 LOC in the core, excluding 1,000 additional lines to pretty-print system calls for policy debugging). Thus, there is little difference in size between the user-level portions of these two systems, or between them and the original Janus prototype, which was just under 3,000 lines of code [36]. The Ostia emulation library is 1,000 lines of additional code, but this is not part of the TCB (trusted computing base) as it runs in the address space of the untrusted application.

The J2 kernel module `mod_janus` consists of 1,400 LOC in C and 11 LOC in *x*86 assembly. The Ostia kernel module `mod_ostia` is only 200 LOC in C and 5 LOC in *x*86 assembly. The difference in complexity in the kernel portion of J2 and Ostia point to significant differences in the impact of each tool on a system's security. A kernel bug would potentially render the entire system vulnerable, as opposed to a bug in a user-space portion which would generally only render the sandbox ineffective.

While a difference of 1,200 LOC may not seem significant, we found the complexity difference between these two modules to be considerable. The difference in development time for these two modules was a few days for `mod_ostia`, versus many weeks for `mod_janus`. To put the complexity of this code into perspective, `ptrace` [27], the standard Linux process tracing interface, consists of less than 300 LOC, offers less functionality than `mod_janus`, and is part of the core kernel which is maintained by experienced kernel developers. Although `ptrace` has been in Linux since version 1.0 or earlier, significant vulnerabilities were found in its implementation during both Linux 2.2.*x* [8] and Linux 2.4.*x* [11] kernel development.

The size of the kernel portions of these systems is still dwarfed by completely in-kernel systems. For example, Subdomain [10] is a relatively small in-kernel solution that restricts access to the file system in a fashion similar to that of our tools. It offers a very simple policy interface, but adds 4,500 lines of code in a kernel module and a patch.

Other security factors: Metrics like lines of code do not tell the whole story on security. Simple code and a conservative design are often far more telling. This is well illustrated by the original Janus system, which although under 3,000 LOC was fraught with security problems, many of which resulted from architectural features that made it particularly prone to race conditions, inconsistent views of system state, and more [16]. Conversely, delegating sandboxes provide an excellent illustration of how system architecture can benefit security.

Delegating sandboxes permit a more conservative design in several ways.

As we noted in section 5.1, delegating sandboxes are relatively easy to render free of race conditions, as most classes of race conditions (inter-process/inter-thread shared state) are eliminated by design due to the fact that the agent performs all sensitive system calls, and inter-process/inter-thread state used by the calls is local to the agent. The remaining potential races are reduced to the much-studied problem of race-free file access by a normal application (i.e. the agent).

Running applications with privilege increases the risk that an application that bypasses the sandbox will be able to inflict damage on the system. Several approaches have been taken that try to mitigate this risk in filtering sandboxes [31]. A delegating sandbox entirely mitigates this risk, because sandboxed processes never run with any privilege. All privilege resides in the agents, as they will be making the system calls requiring privilege on the process's behalf.

If the policy engine contains a bug, it could potentially allow the sandbox to be bypassed, thus it is critical that this portion of the system be as simple as possible. A delegating sandbox like Ostia can simplify its policy engine greatly, as well as other portions of its implementation, by pushing some of the complexity of its TCB into untrusted code, i.e. into the address space of the sandboxed process via the emulation library. For example, the emulation library can reduce the policy engine's complexity by translating operations in the sandboxed process into equivalent sequences for the agent, e.g. if a stub translates `truncate` into the equivalent sequence `open`, `ftruncate`, `close`, then the agent does not need to implement `truncate` at all. Techniques like this account for the 50% smaller size of Ostia's policy engine. The complexity of marshaling the arguments of system calls made by a sandboxed process can also be pushed into the emulation library. This is another example of offloading complexity into the emulation library. In this case, the agent only has to check that arguments, once received, are correctly formatted.

5.2 Flexibility

A sandboxing architecture should be flexible and extensible enough to implement a wide range of security policies and helpful features. Delegating architectures offer expanded potential for easily supporting novel policies. More specifically, because Ostia handles granting access to resources at user level, it is inherently easier to alter the implementation of system calls. This added flexibility can be a boon in a variety of scenarios.

For example, applications can be given their own personalized view of the file system namespace, e.g. mapping `/etc/shadow` to an application-specific copy of the password file, at the same time preserving compatibility and not exposing sensitive state [2]. It can also be useful for mitigating side effects caused by denying system calls [31, 16]. The capability to selectively modify some calls makes it easier to apply other security mechanisms. For example, the sandbox might respond to a request for a socket with a socket running over a SOCKS connection to a firewall, or with a socket that has had a socket filter applied to it. (A socket filter is a Linux primitive for applying fine grain restrictions on what can pass over a socket.)

Filtering sandboxes have provided some support for changing call implementation. For example, some support the ability to rewrite arguments, to change a call's return value, or to change the privilege level of a process while it executes a system call [2, 31]. Unfortunately, for each new change to how a call is executed, new support must be added piecemeal to the kernel. Delegating sandboxes are easily able to accommodate all of these features entirely at user level because the agent controls the execution environment of the call (e.g. call arguments, privilege level, descriptor space) and the choice of calls executed for a given request.

5.3 Compatibility

For greatest utility, an application sandbox must be compatible with a wide range of software. As a first step, it must not require applications to be recompiled or otherwise modified to run them in the sandbox. Ostia, as well as many filtering sandboxes, meets this criterion.

Ostia also supports multithreaded applications. No current filtering sandboxing system supports multithreaded applications due to the problem of race conditions. While it is certainly possible to add this functionality to a filtering sandbox, the significant additional effort and questions of assurance raised have prevented us and others from including this functionality.

Currently our system has been successfully used to sandbox a wide variety of real world applications, including the following:

Network servers: Apache, BIND, CUPS.
Network clients: konqueror, lynx, links, ssh, wget.
X applications: gimp, gphoto, konsole.
Viewers: gs, gv, xpdf, xli.
Editors: Emacs, nvi, vim.
Shells: bash, tcsh.

The broad applicability of interposition-based sandboxing in a practical setting has also been demonstrated by other systems [31, 2].

5.4 Deployability

The fewer prerequisites and dependencies a program has, the more easily that program can be deployed in real systems.

Ostia relies on a kernel module, which causes some deployment difficulty in itself, since it requires that any machine where it is installed has a C compiler and appropriate headers, or a suitable precompiled module. Ostia's module is extremely simple, with few dependencies on kernel internals, which makes porting and maintenance easy. On the other hand, kernel modules required by many filtering sandboxes, including J2, are larger and more complex, with important dependencies on kernel internals that may need careful changes as the kernel evolves, which makes porting and maintenance significantly more difficult.

Ostia does not require a kernel patch, greatly easing the burden on the installer. Applying a kernel patch requires a new system kernel to be compiled and installed followed by a system reboot. This additional human overhead as well as system downtime makes this approach a real impediment to practical adoption. Some systems are loath to be taken down for a kernel patch, normal users often do not have the maturity to comfortably patch and recompile their own kernel, and often even experienced users simply do not want to expend this effort to try a new tool. (Some filtering sandboxes, including J2, also do not require a kernel patch.)

Ostia's loader program must intimately understand the system executable format. A change in the executable format would require modifications to the loader program. Executable formats rarely change, so this is unlikely to be a real barrier to deployment.

In conclusion, we believe that a delegating sandbox such as Ostia can be more easily ported and deployed on a wide range of platforms due to its minimal requirements for kernel support and ease of installation.

5.5 Performance

Architectural features impact performance in important ways. This section undertakes a quantitative comparison of these features, examining the performance impacts of different interposition mechanisms, concurrency strategies, and the overhead of sandboxing on different workloads. We primarily compare Ostia against J2, although other application sandboxes are briefly considered.

Test platform: All of our performance testing was conducted on an IBM T30 laptop with a 1.8 GHz Pentium 4 processor and 1 GB of RAM, running Debian GNU/Linux "sid" "unstable" with a Linux 2.4.20 kernel. Testing was performed in single-user mode with all system services turned off and the network interface disabled. Network service tests were conducted locally over the loopback interface.

Interposition overhead: Table 1 shows per-call interposition cost, the primary overhead imposed by sandboxing. The table's first row shows the basic speed of interposition in each system, using `geteuid`, a trivial system call. On our test system, the minimum penalty of interposition for a system call is therefore about 11.4 μs under Ostia. The second row shows the speed of interposition for `open`, a more substantial system call. Neither J2 nor Ostia has been heavily optimized. In spite of this, its performance for `open` is substantially better than previous published results, which put its slowdown at 25× (25 times slower) [31], compared with our numbers which only reflect 5× to 8× slowdown.

In principle, there are some basic limits on how much this overhead can be reduced. Ostia and J2 both require context switches to and from the policy-checking process for every call they check. This imposes a basic penalty of two additional context switches (essentially one system call) for each checked call in addition to the overhead for making a policy decision.

In a delegating sandbox, some additional calls may be required to obtain a requested resource. Ostia's callback mechanism also requires two additional context switches for the first instance of each type of call it intercepts. This upfront overhead is quickly amortized away over the lifetime of the program. Other overheads, such as the cost of copying arguments, can be kept to a minimum in a careful implementation. This said, we believe that significant further speedups are achievable over our current naive implementation. However, as we will see later, Amdahl's law will likely make further optimizations irrelevant for most workloads.

	sandbox				
call	none	J2		Ostia	
`geteuid`	1.00μs	9.70μs	9.7×	12.42μs	12.4×
`open`	3.92μs	20.42μs	5.2×	31.16μs	7.9×

Table 1: Microbenchmark results. Entries for J2 and Ostia show absolute times and number of times slower than unsandboxed times. Times are "wall clock" times averaged over 10 sets of 100,000 iterations. Over the 10 sets, $\sigma^2 < .15\ \mu$s.

cause	J2	Ostia
`open`	3.92 μs	3.92 μs
basic interposition	8.70 μs	11.42 μs
policy decision	3.27 μs	9.26 μs
extra kernel overhead	4.53 μs	6.56 μs
total	20.42 μs	31.16 μs

Table 2: Time to execute `open` under J2 and Ostia, broken down into individual components: the `open` itself, basic system call interposition overhead, time to make a policy decision, and additional overhead in the kernel.

Where the time goes: Table 2 breaks down the costs of restricting a single open call. We attribute the same cost to the actual file open, 3.92 μs, as in the unsandboxed timings for `open`. We also assume that the basic cost of interposing on a call is unchanged from that for `geteuid`. We calculate the cost of making a policy decision by repeating the measurements with the policy engine turned off and computing the difference. Finally, we assume that the remainder of the time is taken up in additional kernel overhead for checking buffers for file names, copying data, transferring file descriptors between processes, etc., all costs necessitated by `open` but not by `geteuid`.

The table shows that Ostia's policy engine is slower than J2's. This is understandable because the policy engine in Ostia often makes several system calls, whereas the J2 policy engine for file system operations is largely a string-matching operation. The table also shows that Ostia has higher "extra" kernel overhead, which may be due to interprocess file descriptor passing.

benchmark	sandbox none	J2		Ostia	
web serving	10.85s	10.88s	.2%	10.90s	.5%
decompress	3.13s	3.13s	.0%	3.13s	.0%
encode	14.91s	14.94s	.2%	14.92s	.0%
build	8.12s	8.78s	8.1%	10.11s	24.5%

Table 3: Macrobenchmark results. Entries for J2 and Ostia show absolute times and percent slower than unsandboxed times. Times are averaged "wall clock" times. Most entries are averaged over 10 runs with $\sigma^2 < .1$ s; web service entries averaged over 100 runs with $\sigma^2 < .5$ s.

No. Procs.	none	J2	Ostia
1	3.90 s	20.89 s	31.07 s
10	3.94 s	22.79 s	31.62 s
25	3.92 s	29.12 s	32.71 s
50	3.91 s	55.48 s	32.77 s
100	3.91 s	375.96 s	31.87 s

Table 4: Scaling results. Times are "wall clock" time, in seconds, required to open and close 1,000,000 files. "No. Procs" is the number of processes that the file operations were divided among on each row. Entries are the average of 3 runs after an initial, discarded run. $\sigma^2 < .1$ s except for J2 column.

Concurrency scaling: Concurrency strategy can significantly impact scalability, as a lack of parallelism in the monitor or agent can cause a backlog of calls waiting to be checked. Our sandboxes are at opposite ends of the concurrency spectrum: Ostia uses a purely multithreading model, with one agent process per sandboxed process, whereas J2 multiplexes requests through a single monitor process.

To clearly show how a single process can act as a bottleneck, we ran a simple microbenchmark that repeatedly opened and closed files, dividing this work evenly among a variable number of processes. Table 4 shows the results. With no sandbox, running time varied only 1% between 1 and 100 processes; with Ostia, our multithreading sandbox, only 5%. Under J2, our multiplexing sandbox, running time for 100 processes was about 18× that for a single process, and even at 10 processes a 10% increase was observed.

We draw two conclusions. First, the lack of parallelism in a multiplexing sandbox can create a significant performance bottleneck. Even under relatively modest loads this greatly impacts performance. Second, the overhead of naively scaling the number of sandbox processes with the number of application processes is nominal. A third option that we did not explore is a thread pool approach where parallelism could gradually scale with demand. However the added complexity of such an approach seems unwarranted given the success of a naive agent-per-process strategy. Previous work has overlooked the benefits of parallelism, merely citing the overhead of additional processes as the reason for multiplexing [36, 30]. Empirically, multiplexing does not seem to offer any performance benefit; on the contrary it significantly limits scalability.

Typical application overhead: The primary use of sandboxing systems is to protect applications that are routinely exposed to hostile inputs, such as helper applications and network services. We benchmarked three such programs:

Web serving uses Apache to serve 5,000 static pages, totaling 6.4 MB, to a client running outside the sandbox. Pages are requested and serviced serially for this test, so J2's policy engine serialization does not penalize it.

Decompress uses GNU gzip to decompress a 31 MB file, discarding the output.

Encode converts a 48 MB WAV file to Ogg Vorbis format.

The first three rows of Table 3 show the results. In each case, the penalty for sandboxing is less than 1%, because none of these applications uses a great number of sandboxed system calls.

Worst-case application overhead: We also benchmarked a program build. This is an activity not often of interest in sandboxing scenarios. For us, it provides an interesting worst-case benchmark given the large number of restricted system calls performed. It is not entirely contrived, as one might wish to sandbox a build of software downloaded from untrusted locations on the Internet (e.g. to protect against malicious build scripts). Fortunately, building untrusted software is not an activity that takes place frequently, nor does it have real-time requirements as helper applications often do. Thus even the relatively high 25% overhead for this pathological example seems quite tolerable in practice.

Our example build decompresses, unpacks, configures, and compiles the source tree for GNU gzip 1.3.5. This is a system call intensive application, with little CPU needed to compile under 10,000 lines of C, so the cost of sandboxing is significant in the bottom line.

Competing sandbox performance: Table 5 compares J2's and Ostia's performance against published benchmarks of other sandboxing tools. The figures suggest that Ostia performs competitively. The numbers in the table, other than those for J2 and Ostia, are taken from various published sources using different applications and test platforms, so use caution in drawing any more ambitious conclusions.

Class	J2	Ostia	Jain& Sekar	Systrace	MAPbox
network	<1%	<1%	<5%	5%	17%
compute	<1%	<1%	<2%	0%	1%
system call	8%	25%	—	31%	41%

Table 5: Approximate overhead of sandboxing tools on network-intensive (e.g. web serving), compute-intensive (e.g. encoding), and system call intensive (e.g. program build) applications. Numbers for Jain & Sekar obtained from [21], Fig. 6; for Systrace, [31], Fig. 9; and for MAPbox, [2], Table 2.

6 Related work

The first hybrid system call interposition-based application sandbox was Janus, developed by Goldberg [18] et al. Janus set forth the basic architecture for filtering sandboxes. Janus is very similar to J2, as J2 evolved from the original Janus system. Janus initially relied on the Solaris `/proc` interface for interposition. This was noteworthy because it did not require any kernel modifications. An extended description of Janus's architecture is given in Wagner's thesis [36].

The basic Janus architecture was subsequently replicated in MapBox [2], which focused on the problem of policy specification. It was also replicated in consh [3], which leveraged system call interposition to transparently extend the file system and other system interfaces, and to restrict execution. The Systrace [31] system exhibits a filtering architecture very similar to J2. It provides a rich set of features for specifying and generating policy. Systrace stands out as being the most mature and significantly deployed system of this type.

System call interposition has also been a popular mechanism for implementing intrusion detection systems. Several notable examples of this include work by Wespi et al. [39] and Hofmeyer et al. [20].

Jain et al. [21] presented a generalized framework for building secure interposition systems on top of standard process tracing mechanisms. As with all the aforementioned systems, this toolkit had a variety of security problems. Garfinkel [16] presented a full study of potential security problems in these tools, including race conditions, indirect paths to resources, and side effects of denying system calls.

The callback support added to the Linux kernel to support Ostia is reminiscent of a similar feature provided by the Mach system call interception mechanism [25], which redirects system calls to handlers in the same address space as the calling process. Jones' work on interposition agents [22] presents a general framework that provides OS extensibility by placing code in these handlers.

Using this mechanism to convert native system calls into IPC messages to user-level processes via an emulation library, as done in Ostia, is reminiscent of traditional techniques for building Unix emulation layers on top of microkernels. For example, an implementation of 4.3 BSD on top of Mach based on this technique is given by Golub et al. [19]. In contrast to these techniques which virtualize the entire OS interface, Ostia only virtualizes the access control relevant portions of OS API.

Specialized kernel support for interposing on OS interfaces for extensibility purposes has been explored in other work, such as pseudo-devices and pseudo-file systems in Sprite [38] and work by Bershad et al. with Watchdogs [6].

A variety of purely kernel-level [12, 15, 10, 5, 13, 4] and purely user-level [14, 23, 32] sandboxing systems have been presented in the literature. A good comparative survey of sandboxing mechanisms and alternatives, such as whole-system access controls (e.g. DTE [37]), is given by Peterson et al. [30].

7 Conclusion

We have explored the importance of system architecture in secure interposition systems. We presented two systems that implement different hybrid architectures: J2, based on a "filtering" architecture representative of many of today's sandboxing systems, and Ostia, based on a novel "delegating" architecture. We have observed that many of the problems in today's filtering architectures can be ameliorated by a delegation-based approach. Further, a delegating approach can enhance the beneficial properties of existing hybrid approaches.

8 Acknowledgments

J2 was developed in collaboration with David Wagner, who provided invaluable feedback and encouragement during the early stages of this work. Steve Gribble, Costa Sapunzakis, and Ulfar Erlingson also provided helpful feedback at various stages of this work. Cristen Torrey provided valuable editorial assistance. Finally, we are very grateful for the generous feedback and comments of our anonymous reviewers. This work was supported in part by the National Science Foundation under Grant No. 0121481, a Stanford Graduate Fellowship, and by NSF CAREER CCR-0093337.

References

[1] Subterfugue: strace meets expect. `http://subterfugue.org/`.

[2] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proc. 9th USENIX Security Symposium*, Aug. 2000.

[3] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations. `http://www.cs.ucsb.edu/berto/papers/99-usenix-consh.ps`, 1998.

[4] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the Winter USENIX Conference*, 1995.

[5] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Remus: a security-enhanced operating system. *ACM Trans. Information and System Security (TISSEC)*, 5(1):36–61, 2002.

[6] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX file system. In *USENIX Conference Proceedings*, pages 267–75, Dallas, TX, Winter 1988.

[7] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[8] CERT. Vulnerability note VU#176888, Linux kernel contains race condition via ptrace/procfs/execve. may 2002.

[9] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proc. 11th USENIX Security Symposium*, August 2002.

[10] C. Cowan, S. Beattie, G. Kroach-Hartman, C. Pu, P. Wagle, and V. Gligor. Subdomain: Parsimonious server security. In *Proc. Systems Administration Conference*, Dec. 2000.

[11] A. Cox. CAN-2003-0127, Linux kernel ptrace() flaw lets local users gain root privileges. March 2003.

[12] A. Dan, A. Mohindra, R. Ramaswami, and D. Sitaram. Chakravyuha (CV): A sandbox operating system environment for controlled execution of alien code. Technical Report 20742, IBM T.J. Watson Research Center, Sept. 1997.

[13] Entercept Security Technologies. System call interception whitepaper. `http://www.entercept.com/whitepaper/systemcalls/`.

[14] Erlingsson and Schneider. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000.

[15] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

[16] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[17] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proc. USENIX Annual Technical Conference*, pages 39–52, June 1998.

[18] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proc. 6th USENIX Security Symposium*, July 1996.

[19] D. B. Golub, R. W. Dean, A. Forin, and R. F. Rashid. UNIX as an application program. In *USENIX Summer*, pages 87–95, 1990.

[20] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[21] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proc. Network and Distributed Systems Security Symposium*, 2000.

[22] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Symposium on Operating Systems Principles*, pages 80–93, 1993.

[23] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[24] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proc. 9th USENIX Security Symposium*, August 2000.

[25] M. Accetta et al. Mach: A new kernel foundation for UNIX development. In *Proc. USENIX Summer Conference*, 1986.

[26] B. Marick. *lc*. `ftp://ftp.qucis.queensu.ca/pub/software-eng/software/Cmetrics/lc.tar.gz%`.

[27] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*, pages 112–114. Addison-Wesley, 1996.

[28] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Proc. 13th Annual Computer Security Applications Conference*, December 1997.

[29] V. Nakra. Architecture study: Janus—a practical tool for application sandboxing.

[30] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *Proc. 11th USENIX Security Symposium*, August 2002.

[31] N. Provos. Improving host security with system call policies. In *Proc. 12th USENIX Security Symposium*, pages 257–272, august 2003.

[32] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Proc. Annual Computer Security Applications Conference*, 2002.

[33] Steve Bellovin. Shifting the Odds, Writing More Secure Software. `http://www.research.att.com/~smb/talks/odds.ps`.

[34] Teso Security Advisory. LIDS Linux Intrusion Detection System vulnerability. `http://www.team-teso.net/advisories/teso-advisory-012.txt`.

[35] J. Viega and G. McGraw. *Building Secure Software*, pages 209–229. Addison-Wesley, 2002.

[36] D. A. Wagner. Janus: An approach for confinement of untrusted applications. Technical Report CSD-99-1056, University of California, Berkeley, 12, 1999.

[37] K. M. Walker, D. F. S. anad M. L. Badger, M. J. Petkac, D. L. Shermann, and K. A. Oostendorp. Confining root programs with domain and type enforcement. In *Proceedings of the sixth USENIX Security Symposium*, July 1996.

[38] B. Welch and J. Ousterhout. Pseudo devices: User-level extensions to the Sprite file system. In *Summer 1988 USENIX Conference*, pages 37–49, San Francisco, CA, 1988.

[39] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable length audit trail patterns. In *RAID 2000*, pages 110–129, 2000.

A Ostia Policy Interface

Starting State

`starting_uid` *`uid`*
`starting_gid` *`gid`*
`starting_dir` *`directory`*

Sets the initial user id, group id, and current working directory for the sandboxed process.

File System Policy

`path-allow (read|write|unlink|exec)...` *`path...`*

Provides access to file system resources. Files to which access is granted are written as absolute file names that may pattern matching wildcards. Keep in mind that directories are files and must be authorized the same way; e.g. to `stat` a directory, read permission for that directory is required.

Examples:

`path-allow read /var/foo`
Allows the contents of `/var/foo` to be read.

`path-allow read /var/*`
Allows any file whose absolute path begins with the prefix `/var/` to be read.

Network Policy

`net-allow (outgoing|incoming) (tcp|udp)` *`address`*`[/`*`mask`*`]` *`port`*`[/`*`mask`*`]`

The `net-allow` directive controls access to network resources and limits IPC over sockets. All application use of sockets must be explicitly allowed. Creating outgoing connections to other local or remote processes and accepting incoming connections from other processes are controlled separately.

The syntax to allow a sandboxed application to connect to another process or send traffic directly to it is `net allow outgoing` *`type end-point`*.

To allow a sandboxed application to bind a socket (i.e. wait for a connection from some other process) or receive traffic from another address, write
`net-allow incoming` *`type end-point`*

Examples:

`net-allow incoming unix-domain /var/*`
Allows a sandboxed process to bind a socket with any path in /var.

`net-allow outgoing tcp 128.36.31.50 80`
Allows tcp connections to be made to the host at 128.36.31.50 on port 80.

`net-allow outgoing tcp MYHOST_ADDR 0/0`
Allows a sandboxed process to make tcp connections to any local port. Keyword `MYHOST_ADDR` is special syntax for the local IP address.