

iPoJ: User-Space Sandboxing for Linux 2.4

Brian Greskamp

Pablo Montesinos

Paul Sack

i-acoma group

Department of Computer Science

University of Illinois Urbana–Champaign

201 N. Goodwin Ave.

Urbana, IL 61801

E-mail: {greskamp, pmontesi, paulsack}@cs.uiuc.edu

Abstract

The Internet is a dangerous place. Both naïve and educated users routinely become infected with viruses and accidentally run spyware despite widespread knowledge of such risks and how to avoid them. Clearly, user education is a dead-end. Another approach is to finely limit what different applications can do so that viruses and malware can't do any damage in the first place. We observed that system-call execution is needed for viruses and malware to do any damage, so we developed iPoJ, a user-space sandboxing environment for Linux. With this, application and OS vendors can precisely define which system calls each application can execute and how. A properly iPoJ-protected user exposed to a virus or spyware would escape certain death unharmed.

1. Introduction

Untrusted applications are everywhere. The naïve user is vulnerable to trojan horses and spyware included in widely distributed and seemingly trustworthy applications. Even experienced users are vulnerable. Who inspects the source code before compiling and running a free software application? What is needed is a way to run a program in an “untrusted” or “sandboxed” environment, in which the environment enforces fine-grained, highly configurable rules before executing potentially damaging code. This paper describes iPoJ, a user-space sandboxing environment for Linux.

1.1. Malware

Trojan horses, worms, spyware, and viruses damage computer systems and users by disseminating personal or confidential information, destroying or modifying user's files, using system resources, sending spam, and infecting other vulnerable systems.

For example, if a user uses a vulnerable email reader and receives an email infected with the Sobig worm [6], the user's system will become infected. First, the worm copies a binary executable and modifies the Windows registry so that the worm runs every time Windows is started. Sobig then scans files on the user's hard drive and any networked file systems for email addresses and then sends infected emails to the email addresses it finds. Further, Sobig can download a program from a master server and execute it on the host system. The downloaded program runs with full user privileges and can be used for all sorts of malicious purposes.

The MyDoom worm works similarly and also installs a key-logger program used to record passwords and credit card numbers [7]. It also can launch a distributed denial-of-service attack.

What these viruses have in common is that they use *system calls* to wreak havoc¹. Both programs have to *open* files and directories to read them, *execute* payload binaries, and *connect* to Internet sockets to spread themselves, and each of these requires executing a system call through the kernel.

¹And don't infect Linux systems ... but that's not the point!

1.2. Stopping Malware

A generic way to stop malware from spreading is to restrict the system calls that different programs can use, since on an operating system with protected memory, applications can do harm only through system calls. For example, a game could be restricted such that it can no longer access the network or files outside of its data directory. A user running an unknown application for the first time could run the application in a very restrictive sandbox.

A sandbox mechanism that runs entirely in user space is desirable for several reasons; it requires no operating system modification or special privileges to use. It can run as a standalone application or it can be embedded in an existing application. For example, an email client or a web-browser could open up attachments or execute plugins inside a sandbox that the email client or web-browser controls. Policies can easily be specified for different applications and different users. Finally, user-space sandboxing is portable between different operating system versions.

Consequently, if we can restrict the system calls that an application can make, then we can prevent damage. On most UNIX systems, the `ptrace` API allows for intercepting and filtering system calls. Although `ptrace` semantics vary widely from system to system, most UNIX flavors implement the same basic functionality, including signaling a *tracer* process when a *traced* process executes a system call. To make use of this functionality, we create a sandbox process which will be the *tracer* for all processes and threads in the untrusted application.

Since different applications need access to different resources, sandbox policy must be configurable with flexible rules. We accomplish this with a chain of plugins implementing rules for different calls.

In this work we have developed iPoJ, a configurable user-space sandboxing tool that can be used to effect a safer computing environment. The rest of the paper is organized as follows. Section 2 discusses background information and related work. Section 3 discusses our implementation approach. Sections 4 and 5 present an evaluation of our work, and Section 6 concludes.

2. Background

Linux, a UNIX clone, follows the simple security model of UNIX — protecting the system and its users from each other. This is done mostly by giving each process a private virtual memory space and by enforcing file-system permissions. The system is also protected by only allowing the superuser to execute sensitive system calls. This security model does not protect a user from himself — that is, from code that he inadvertently runs.

2.1. Access Control

Many modern UNIX implementations do provide a finer-grain security model including *capabilities* [11] and filesystem *access control lists* (ACLs) [8, 15]. ACLs allow more freedom in specifying file permissions than the standard user-and-group model. Each file carries a list of users that are authorized to access it and what permissions each user has. Capabilities are associated with executable binary files. They specify which system facilities should be available to the program. For example, on Linux, the capability `CAP_NET_RAW` determines whether the program can use the low-level network interface to send non-TCP or non-UDP packets and `CAP_SYS_TIME` controls the ability to set the system clock.

While capabilities and access control lists provide finer grain control, they are still not fine enough, especially when it comes to controlling network access. For example, it is not possible to specify that the mail client should be able to communicate only with `smtp.uiuc.edu` and `pop3.uiuc.edu`. Also, using ACLs requires creating a new user for each set of permissions the user wishes to create. If user `joebob` wants to limit the files that Firefox can access, he creates a new user, `joebob-firefox`, and always runs Firefox under that user. Furthermore, he must add `joebob-firefox` to the ACL for each file that Firefox is allowed to access. This amounts to a lot of work for a partial solution.

2.2. Kernel-space Sandboxing

What we really want is the option to run each application in a sandbox with a unique and fully configurable security policy. One common approach to

building sandboxes is to insert routines in the kernel to intercept system calls and apply filtering.

Janus [9], Systrace [13], Medusa [2], and RS-BAC [3] are sandboxes that have both kernel and user-space components. The kernel component performs interception, and the user-space process performs filtering and signals the kernel module to either allow or reject the call. Doing the interception in kernel space offers freedom from race conditions (as described in Section 2.4), while filtering in user space makes debugging easier and allows the use of tools like PERL and TCL for rule processing. The Linux Security Modules [1] framework provides hooks into over 150 different system events and provides a stable interface on which to build kernel-space sandbox components.

Still, a kernel-space sandbox approach is unattractive to us because it requires the system administrator to install and configure the sandbox. It also will be less portable because it is using less stable kernel interfaces. Finally, it is more difficult to implement and test a kernel sandbox, because kernel programming and debugging is notoriously difficult. An alternative approach that we prefer is to use a user-space sandbox built on the `ptrace` API.

2.3. User-space Sandboxing

It is possible to implement a sandbox program entirely in user space using the POSIX `ptrace` API. Existing programs such as Subterfuge [5], S4G [12], and older versions of Janus [9] work this way. Essentially, `ptrace` allows one process (the sandboxer) to trace another (the sandboxed application). It can be used to suspend the execution of the traced process and inspect and modify its registers, code, and data. Its primary use is in implementing a debugger, but it also has features that allow one to build an efficient sandbox. The Linux implementation has an option wherein the traced process is interrupted before and after system call execution. This allows a sandbox to inspect system call arguments and accept or reject the system call.

Figure 1 shows how the tracer intercepts and filters calls. In step 1, the untrusted process writes the arguments for the system call into registers and memory (according to the ABI). In step 2, the untrusted process invokes the system call (by executing `int 0x80` on

Linux/x86). The operating system then suspends the untrusted process and signals the sandbox process in step 3 (according to `ptrace` semantics). In step 4, the sandbox reads the system call arguments and decides whether the call should proceed. Here the sandbox has three options: (1) allow the system call to proceed by resuming the calling process; (2) deny the system call returning an error to the calling process; or, (3) modify the arguments so that they are acceptable and then resume the calling process. Finally, the result of the system call is returned to the untrusted application in step 5.

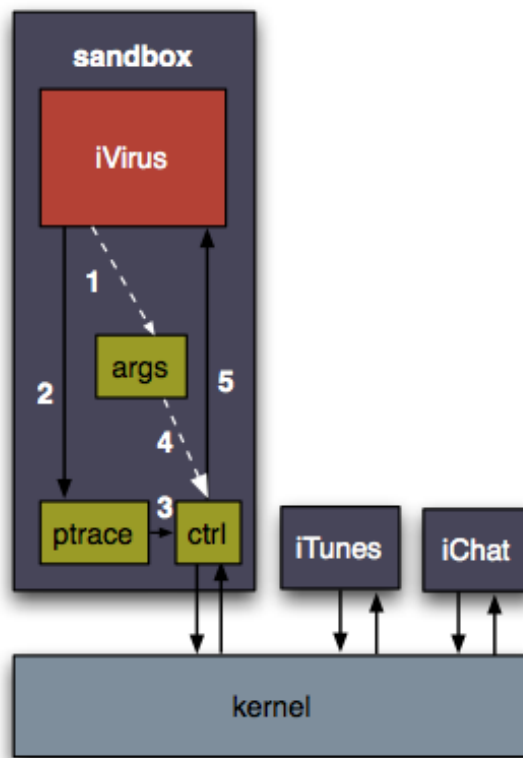


Figure 1. Flow of control between sandbox, untrusted process, and kernel.

2.4. Pitfalls of User-space Sandboxing

While the basic idea of `ptrace`-based userspace sandboxing is simple, we will see that subtle race conditions threaten security.

2.4.1. Multi-threading Argument Race

In a multi-threaded application, a subtle race condition can occur as follows.

1. Thread A tries to make a system call with “safe” parameters.
2. The sandbox intercepts the system call and validates the parameters as safe.
3. Thread B modifies the parameters to the system call to something unsafe. It can do this because threads A & B share the same virtual memory space
4. The sandbox allows the system call to proceed

To be vulnerable, some of the system call parameters must reside in memory and not in registers. Although the kernel system call interface requires all parameters to be in registers, some of these registers may be pointers to memory. The memory might contain a string containing a filename, for example. System calls that use pointers for input arguments may be vulnerable to this attack. We call such system calls *volatile* (see Appendix A for a complete listing). Other system calls that only use scalar arguments are invulnerable to this attack, as threads cannot modify each other’s registers directly.

Proposed Solution When a volatile system call is to be executed, we temporarily suspend all the other threads in the process, check the arguments, allow the call to execute, then resume all the threads.

It would be simpler, albeit slower, to always suspend all the other threads in the process when any system call is executed rather than only when volatile calls are executed. It turns out that such a solution would also be incorrect and could lead to deadlock. Developers often use bizarre forms of synchronization through system calls. For example, [14] documents a common technique of two threads synchronizing through paired blocking `read` and unblocking `write` calls to a shared pipe, or multiple threads synchronizing with one thread through `write` and `select` calls to multiple shared pipes. If we were to suspend all threads in a process group when a blocking `read` or `select` call is executed, the unblocking `write` call would never get to execute, and we would have deadlock.

We are not aware of any pair of matching blocking and unblocking system calls in which one of the calls is volatile, so suspending all other threads only on volatile calls is an effective solution.

2.4.2. Fork–Attach Race

Another race condition is the fork-attach race. After a `fork` system call executes, the sandbox may not immediately attach to the child process. The child process may get to execute for a small amount of time before the sandbox can attach to it and enforce its policies. In this small amount of time, the child process could do “malicious things” such as fork off another process that the sandbox will never know about.

Proposed Solution Our solution, and that used in other work, is to change the call from `fork` to `clone`. The `clone` call has an option to start the child process in a `ptraced-mode`. The child processes registers are twiddled appropriately to make this operation transparent.

2.4.3. Filesystem Race

This is the trickiest race to solve. There is a class of attacks that uses symbolic links to break security. The race works as follows.

1. The sandboxed process tries to do a filesystem system call that involves a path.
2. The sandbox looks up the path, follows all the symbolic links until an absolute path is acquired, and verifies the path.
3. Another process, or another thread in the same process replaces one of the components in the path with a symbolic link to point to a secure region in the filesystem.
4. The sandbox approves the system call, which now uses an arbitrary unsafe pathname.

Proposed Solution We resolve all the symbolic links in a path before verifying it. We temporarily suspend all other process threads before allowing the system call to proceed, so that no other threads can create symlinks until the system call completes.

Our security model is not intended to protect an application from other malicious applications, so we ignore the possibility of an outside application manipulating the file-system to execute an attack.

From our reading, it seems that the only solution to this is to either run all user processes in a sandbox, or use a kernel-space solution. Running all user processes in a sandbox would require heavy system modification, and the sandbox would clearly need superuser privileges. For example, the system would have to be reworked such that all user logins and scheduled jobs executed under the sandbox. Sshd, crond, etc., would all need to be modified.

2.5. Security Model

Our system is designed to sandbox individual user process groups. The intention is that it could be used for sandboxing individual programs that might be vulnerable to attacks, e.g., web browsers, email clients, macro viruses, etc. Clearly, different applications should be given different privileges, which is simple when each application has its own sandbox. When a process is executed under iPoJ, the rules apply transitively to that application and all of its children.

From our literature survey, it is clear that grand projects that attempt to sandbox the entire system, or at least all user processes together, are difficult to implement correctly. Many projects are vulnerable to subtle race conditions and others become difficult to install and configure, defeating the purpose. Ours is simple enough that it could actually be used.

3. Implementation

iPoJ is implemented in C++ and is designed to work with the Linux 2.4 system call interface. This section describes the implementation details, starting with a primer on `ptrace` and the Linux system call interface. We then present the mechanisms for overcoming the fork-exec and argument-modification races. It is important to note that even though iPoJ is a userspace program, the system call and `ptrace` interfaces are system dependent. The rest of this discussion presumes Linux 2.4 on an x86 processor.

3.1. Compatibility

iPoJ provides extrinsic compatibility only. No sandboxed application is allowed to call `ptrace`, as it is not clear how “nested” tracing works. Allowing child processes to access this facility might enable them to escape the sandbox. This restriction precludes nesting iPoJ sessions or running a debugger in a sandbox. More subtly, it affords a malicious program a means of determining whether it is running under a sandbox; `ptrace` calls will fail where they normally would not. A program may thus suppress malicious behavior to avoid raising suspicion when run under the sandbox. However, the mere attempt at using `ptrace` is enough to reveal it as a possible threat.

Compatibility is also broken for applications that use the `SIGPWR` signal, since iPoJ uses this signal internally. Although `SIGPWR` was originally intended to indicate a power supply failure, infrequently-used signals are in high demand and are being reclaimed *ad hoc* by application programmers. It is used, for example, in the GCJ runtime, and thus GCJ applications will not run under iPoJ. A simple workaround would be to allow the user to select on the command line the signal number (default = `SIGPWR`) that iPoJ will use internally.

Nevertheless, the majority of applications *are* compatible with iPoJ. We tested iPoJ with OpenOffice, Firefox, Thunderbird, Evolution, Bash, NEdit, and several other applications. Among these, we found no compatibility or stability problems.

3.2. `ptrace` and Syscall Primer

Since our system call interposition method is based on `ptrace`, it is important to understand the capabilities that `ptrace` does (and does not) provide. Most of GDB’s functionality, except for breakpoints², comes from `ptrace` calls. For our purposes, `ptrace` provides three key operations: (1) Receive notification when a process enters or leaves a system call or receives a signal; (2) Get or set the register context for a process; (3) Get or set bytes in the traced process’s text or data segment. In (1), the tracing process, our sandbox, is notified via `wait()` when an event occurs in a

²Breakpoints use code modification, replacing the target instruction with `int 0x3`

traced process. The traced process is then suspended, allowing the sandbox to use operations (2) or (3) to inspect and manipulate its state before allowing it to resume.

With `ptrace`, creating a sandbox for single-threaded applications is straightforward. To treat multi-thread and multi-process applications, one needs to understand some kernel-level details. First, threads are kernel-level entities and are scheduled just like processes. In fact, the same kernel function, `do_fork`, is responsible for creating both. Threads and processes alike are identified by a unique TID (task identifier). For processes, the TID and the familiar UNIX PID are the same number. From now on, we will use “TID” and “task” where the entity could be a thread or a process and “PID” or “process” when it is definitely a process.

For iPoJ, the important difference between threads and processes is that threads have different signal-handling semantics. Each thread is associated with a *thread group*. All threads created by a given process are in that process’s thread group, and the creating process is the *leader*. When a signal is sent to a TID via `kill`, it can be delivered to *any* thread in the group and is guaranteed to also be delivered to the leader PID. This poorly documented behavior caused endless frustration, since iPoJ occasionally needs to signal a specific TID. Kernel 2.4.20 introduced the `tkill` call, which does precisely that. A signal sent with `tkill` goes to the specified TID and to its leader PID. It is because of the dependence on `tkill` that iPoJ requires kernel version 2.4.20 or higher.

3.3. Fork Coercion

All UNIX-like systems support the `fork` call, which creates a new process by logically copying the current process image. Linux also implements a more flexible `clone` call, which creates a new task that optionally inherits memory regions, open files, signal masks, and `ptrace` status from its parent. It is therefore possible to “emulate” `fork` with `clone`.

When we replace `fork` with the appropriate `clone` and request that the child inherit the parent’s `ptrace` status, we sidestep the fork–attach race. If the parent was in the sandbox, then it was being traced and tracing of the child will commence immediately. The

challenge then is to *coerce* the `fork` call to a `clone` in a way that is transparent to the application.

Linux passes all system call arguments in registers. Whereas `fork` takes no arguments, `clone` takes five. This means that at the point of a `fork` call in an application, all of the registers that will be needed to hold the `clone` arguments may be live. Consequently, it is necessary to save the register context of the caller so that the `clone` arguments can be written. On return from the call, both in the parent and in the newly created child, the sandbox must restore the saved state. Failure to restore the registers to their values from before the coercion would result in corruption of any registers that were live at the time of the `fork` call. The only register not overwritten during restoration is `%eax`, which holds the system call return value. The return codes for `fork` and `clone` are identical, so no translation is required.

The only real challenge is how to find the correct saved state for each new child. The core of the problem is that on Linux 2.4, there is no easy way to determine which task created another. So, as the newly created task emerges from `clone`, we must somehow determine who created it so that we can restore the pre-coercion state from the creator. One possibility is to allow only one outstanding `clone` at a time. In this case, the sandbox saves the state in a global location on entry to `clone`, and the child claims it on exit from `clone`. As simple as this solution seems, ensuring a single outstanding `clone` would complicate the control logic. Instead, the sandbox allows an arbitrary number of in-flight `clone` calls and uses a free register to pass the child a pointer to an in-memory copy of the saved registers. The child then uses the pointer to grab the saved register image, restore it, and free the associated memory.

In the register-starved x86 architecture, finding a register in which to store a pointer to the saved register image is a problem. All of the integer registers are pressed into service in the ABI for passing `clone` arguments, so we store the pointer at the top of the floating point register stack. Since cloned children inherit the floating point as well as integer register values from their parent, this works fine; the newly cloned child has the pointer at the top of its floating point register stack as it emerges from `clone`. The only disadvantage is that iPoJ only works on systems with a hardware FPU.

3.4. Avoiding Data Races

Section 2 described the idea of *volatility*. When iPoJ receives a system call entry event from one of its children, it determines whether the request is volatile through a lookup on the syscall number, but it does not filter the call right away. Instead, it places the TID of the requester into one of two sets: **N** if the request is nonvolatile or **V** if it is volatile. In addition, iPoJ maintains the set **U** of all tasks in the sandbox. After each event, the sandbox invokes a state machine that examines both sets to determine which call should be processed next.

Figure 2 depicts the iPoJ state machine. When in the nonvolatile state, requests from the **N** set are retrieved, filtered, and allowed to proceed concurrently. In this state, thread level parallelism is fully exploited on a multiprocessor machine; the only point of serialization is the filtering itself.

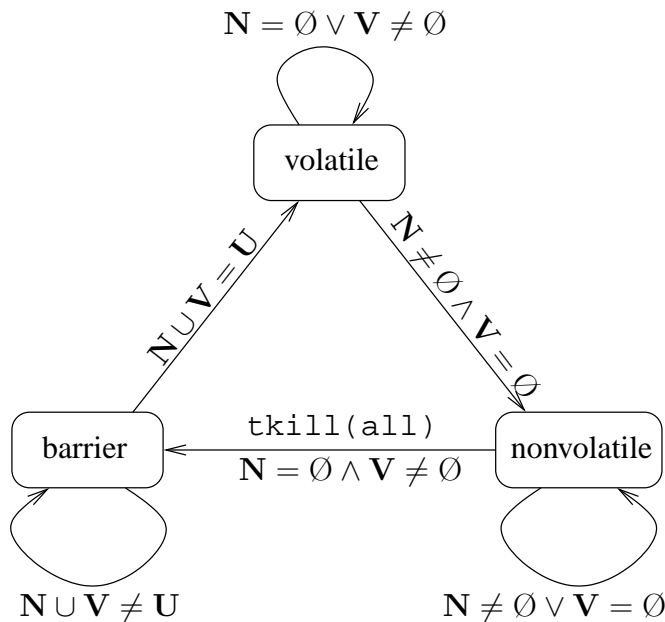


Figure 2. Transitions between volatile and normal execution modes. The sets **V**, **N**, and **U** represent tasks with volatile requests, tasks with nonvolatile requests, and all tasks, respectively.

When **N** becomes empty, the sandbox must begin filtering volatile requests if any are waiting. In this case, it must cause all tasks to stop before the first

volatile request can proceed. This is achieved by sending a special signal (`SIGPWR`) to each task in **U** and then transitioning to the “barrier” state. When a traced task receives a signal, the kernel stops it and notifies the tracer via `wait`. In the barrier state, the sandbox waits until all tasks have stopped. To ensure that all tasks will eventually respond to `SIGPWR` and stop, iPoJ intercepts calls to all functions that can change signal masks or handlers and ensures that nobody interferes with the sandbox’s use of `SIGPWR`.

When all tasks are stopped, the sandbox enters the “volatile” state. Here, it selects and filters a syscall from **V** and waits for the call to return before filtering the next volatile request. Consequently, volatile tasks execute exclusively of all other tasks. Once the **V** request pool is empty and a **N** request arrives, the sandbox transitions back to the “nonvolatile” state and resumes concurrent execution. Note that the transition logic is designed to minimize the number of barrier operations, which have time complexity = $O(n)$ in the number of tasks.

3.5. Rule Infrastructure

iPoJ implements a simple chain-based rule structure, where rules are implemented as dynamically loadable libraries (`.so` files). The user sets an environment variable, `IPOJ_RULES`, before starting iPoJ. The variable contains a list of colon-delimited module names. Each module provides a `check_rule()` method that returns a boolean, `true` if the rule allows the call or `false` if it does not. The register context is passed to the rule, and the rule can use additional `ptrace` functions to examine the task’s memory space. We also provide helper functions that use the `ptrace` API to copy common data types from the task’s memory space. If all of the rules in the chain return `true`, then the sandbox allows the call to proceed. If one of the rules returns `false`, then the system call is aborted.

To abort a call, the sandbox first converts the call to an innocuous `getpid` call by changing the value in `%eax`. It then sets the state of the child to “aborting”. On return from `getpid`, the sandbox places the `-EPERM` return code in `%eax`. To the user application, it appears as though the denied call suffered a standard UNIX “permission denied” failure. The child process

continues running and handles the error normally. Optionally, the rule may also print a warning to the user alerting him of suspicious behavior. A sample rule for filtering `open` calls with regular expressions is shown in Appendix B.

3.6. Unimplemented Features

Some additional work would be required before a full-featured version of iPoJ is ready for release. First in the list of missing features is *path resolution*. When a filename is passed to a system call, the file often contains relative paths (e.g. "`foo`" or "`../foo`"). Additionally, paths may contain symbolic links. To filter filesystem operations, it is necessary to resolve the full absolute path for each of the operation's arguments. Currently, all of the rule modules see only relative paths, and we must forbid `chdir` and `fchdir` if we wish to filter file names.

Two possible solutions are as follows: (1) Let the sandbox track the current working directory of each child. Upon intercepting a filesystem operation, it uses the working directory to resolve the full path. (2) Before each filesystem operation, force the *child* to execute the `realpath()` library function on each argument. While (1) is a simple solution, it involves mirroring state (the current working directory) in the sandbox, and incorrectly mirroring state is a major source of security holes [9]. Even though it seems that only two system calls (`chdir` and `fchdir`) can change the working directory, this is difficult to prove.

To illustrate the subtleties associated with mirrored state, consider the following example. Two processes *P* and *Q* are executing in the sandbox *S*. The current working directory is `"/P"` for *P* and `"/Q"` for *Q*. Next, *Q* removes `"/P"` and creates a new directory by the same name. It then adds a file "`bar`" to the directory. Finally, *P* does an operation on "`bar`". The sandbox might interpret this as an access to the same `"/P/bar"` that *Q* created, but this is incorrect! There is no file named "`bar`" in *Q*'s version of `"/P"`. It is not immediately clear whether this can be a security threat, but it does demonstrate the possibility of unanticipated behavior.

Solution (2) then is vastly preferred from a security standpoint. It could be implemented with binary modification as follows. After encountering a system call

with paths that need to be resolved, the sandbox could replace the system call with a call to `realpath()`, abort the system call, rewind the program counter by one instruction, and resume the task. The task would then execute `realpath()` in its own context and store the result in a memory segment shared with the sandbox. After completion of `realpath`, the sandbox would replace the original code for the system call, adjust the program counter, and resume the child once again. The child would then re-execute the system call and the sandbox could inspect the resolved paths stored in the shared memory segment. While secure, this technique obviously involves a large amount of overhead (due to invalidating the I-cache) and implementation effort.

4. Experimental Setup

Evaluating iPoJ is not an easy task. We cannot evaluate iPoJ with benchmarks like SPECint or SPECfp because they use very few system calls. Moreover, iPoJ is targeted to the average desktop user and, consequently, a 10% slowdown in *mcf* would not be too meaningful for him. Therefore, we created three benchmarks that try to capture what a desktop user does and, at the same time, stress iPoJ in different ways:

- **FoxBench:** FoxBench uses Firefox to browse through fifty websites. Firefox waits ten seconds on each page before loading the next. The pages are a mix of simple pages, pages with lots of animations and Flash, secure pages and pages with lots of images. FoxBench is executed five times and each run goes through the list of pages five times, cleaning the cache only for the first run of each execution.
- **TunesBench:** One of the most common uses of desktop computers nowadays is to listen to *mp3* music. TunesBench converts a one hour long, 44 khz, 160 kbps *mp3* file into a wav file. It hides hard drive latency by warming up the disk cache before the first run and by writing its output to `/dev/null`. This process is repeated 20 times.
- **SpotBench:** SpotBench uses the UNIX `find` utility to search for file names matching a regular expression. SpotBench traverses a directory

containing about fifty thousand files five times per run, looking for a different file each time. We execute SpotBench fifteen times.

- **WorkBench:** WorkBench opens a complex office document in OpenOffice and measures the startup time. It repeats the experiment 15 times and discards the first run.

We run iPoJ with two rule modules—one disallows the `unlink` call entirely, and another intercepts `open` calls and allows or disallows them based upon the type of access required (read, write, read/write) and a regular expression over the filename. This rule requires copying the string containing the pathname from the traced application using `ptrace` calls.

5. Evaluation

5.1. System Call Interface Usage

To evaluate iPoJ, we first analyze how the different benchmarks use the system call interface and then we evaluate iPoJ’s performance.

Table 1 shows, for each benchmark, its five most used system calls. Volatile system calls are shown in bold-face. We can see that the five most-used system calls in FoxBench are nonvolatile, so we should expect a very small overhead from thread-suspension in multiprocessor systems. Another interesting aspect of FoxBench is that half of the calls are to `gettimeofday`. The reason for this is that FoxBench is an X application and X applications call `gettimeofday` very frequently to measure the time between UI events. Notice that this does not represent a problem for iPoJ as `gettimeofday` is nonvolatile and it does not need to execute exclusively.

On the other hand, three of the five most-used system calls in SpotBench are volatile. Since SpotBench is a single-threaded application, iPoJ will not need to suspend other threads when handling volatile calls.

TunesBench is interesting because almost all of its system calls are reads. The `read` call is nonvolatile; in fact, it requires no filtering at all. Since the call requires an open and readable file descriptor to specify the file to read from, we can rely solely on open filtering. In other words, any valid descriptor that is passed to `read` was at some point initialized by an

open or connect, which *was* filtered. Although it requires no filtering, `read` is called very frequently. As a result, TunesBench is likely to experience slow-downs because of the overhead introduced by iPoJ in the system call interception process. Finally, WorkBench’s behavior is similar to FoxBench.

5.2. Performance

We consider two aspects when we evaluate the iPoJ performance: wall-clock execution time and total CPU time. Figure 3 shows the normalized wall-clock execution times and Figure 4 breaks down the CPU time of each benchmark into system and user time.

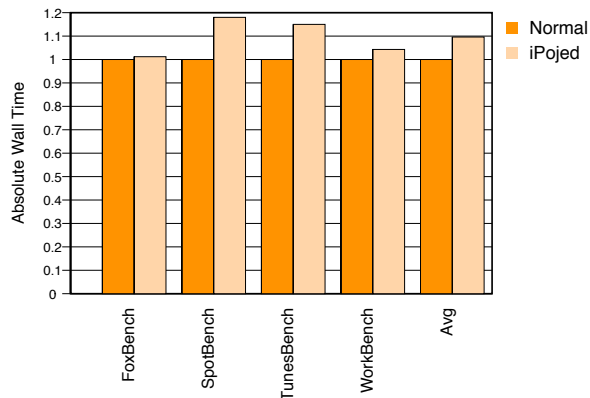


Figure 3. Normalized execution time for normal and sandboxed executions.

As expected, a sandboxed FoxBench has only a 1.2% wall-clock execution time overhead. Even though both system and user CPU time increase, network delays hide iPoJ’s overhead and, therefore, the user will not notice any difference. The other benchmarks do not behave as well.

More than 60% of the system calls executed in SpotBench are volatile. This, however, is not the reason why SpotBench runs over twice as slow under iPoJ, as SpotBench is a single-threaded benchmark. The real reason is that the frequently executed `open` calls generate several extra `ptrace` calls under iPoJ. Our rule for validating `open` calls executes several `ptrace` calls to read the filename parameter in 4-byte chunks. Each of these calls involves a context-switch from user-space to kernel-space and back. Validating a small 32-byte

| FoxBench | | | SpotBench | | | TunesBench | | | WorkBench | | |
|--------------|--------|-------|----------------|-------|-------|-------------|--------|---------|--------------|------|--------|
| S.C. | # | % | S.C. | # | % | S.C. | # | % | S.C. | # | % |
| gettimeofday | 212867 | 49.09 | lstat64 | 20071 | 40.99 | read | 214706 | 99.9667 | read | 8491 | 23.76 |
| read | 38194 | 8.81 | chdir | 7048 | 14.22 | open | 31 | 0.004 | gettimeofday | 4569 | 12.78 |
| futex | 35774 | 8.25 | getdents64 | 6941 | 14.11 | stat | 8 | 0.003 | lseek | 3496 | 9.7834 |
| write | 32254 | 7.43 | open | 3492 | 7.12 | brk | 7 | 0.003 | write | 3032 | 8.48 |
| poll | 30223 | 6.97 | fcntl64 | 3471 | 7.08 | write | 5 | 0.002 | futex | 2747 | 7.69 |

Table 1. Five most used system calls of each benchmark and how many times they were called. Volatile calls are in bold.

filename requires 16 extra switches between user and kernel mode, and most of the filenames handled in SpotBench are much longer than 32 bytes. Filters which validate parameters in volatile calls will usually require copying data from user applications and incur similarly high overheads.

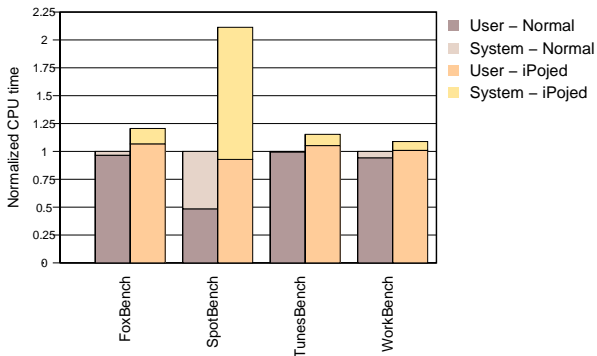


Figure 4. Normalized user and system CPU time for normal and sandboxed executions.

TunesBench runs 14% slower under iPoJ. Unlike SpotBench, where the overhead comes from parsing the system call arguments, TunesBench suffers slowdowns from repeatedly executing the `read` system call. Although `gettimeofday` in FoxBench is called as many times as `read` in TunesBench, FoxBench runs for a much longer time and can use the idle cycles from the network delay to hide the overhead. Because TunesBench spends very little time in kernel mode when executed outside iPoJ, Figure 4 shows that most of the overhead comes from system CPU time.

What we observe is that applications that execute mostly non-volatile system calls suffer modest slow-

downs. Applications that execute mostly volatile calls suffer heavy slowdowns, since rules used to verify volatile calls are very slow.

6. Conclusion

Viruses, worms, bacteria, and parasites are the scourge of computer users everywhere. For every worm and virus out there, there is a grande project available in alpha form claiming to provide a Teflon™ shield protecting the naïve user from anything and everything.

We have presented iPoJ, a user-space sandboxing environment for Linux. It uses the existing `ptrace` API and so does not require any kernel modification. Most of all, it is simple and practical, and, therefore, easy to verify.

We have demonstrated how avoid common sandboxing pitfalls. We have shown that the performance impact is unnoticeable to the common user. iPoJ won't solve all user problems — we can do little to fix broken backlights or frayed Firewire cables — but iPoJ can make life a little safer for all the Internet users out there.

A. Volatile Calls

Out of 246 system calls in Linux 2.4.20, only the following 45 are volatile: `open`, `creat`, `link`, `unlink`, `chdir`, `mknod`, `chmod`, `lchown`, `mount`, `umount`, `utime`, `access`, `rename`, `mkdir`, `rmdir`, `umount2`, `chroot`, `sigsuspend`, `sigpending`, `symlink`, `readlink`, `uselib`, `swapon`, `truncate`, `statfs`, `stat`, `lstat`, `swapoff`, `._sysctl`, `chown`, `truncate64`, `stat64`, `lstat64`, `lchown32`, `chown32`, `pivot_root`, `setxattr`,

```
lsetxattr, getxattr, lgetxattr,
listxattr, llistxattr, removexattr,
lremovexattr, socketcall
```

B. Sample Rule

```
int check_rule(int syscall_num, pid_t pid,
               struct context* const state)
{
    int allow = 1;
    if (syscall_num == __NR_open) {
        char fn[256];
        if (copy_n_string(fn, sizeof(fn),
                          pid, state->iregs.ebx) < sizeof(fn)) {
            switch(state->iregs.ecx & 0x3) {
                case O_RDONLY:
                    if (regexec(&read_rx, fn, 0, NULL, 0) != 0)
                        allow = 0;
                    break;
                case O_WRONLY:
                    if (regexec(&write_rx, fn, 0, NULL, 0) != 0)
                        allow = 0;
                    break;
                case O_RDWR:
                    if (regexec(&read_rx, fn, 0, NULL, 0) != 0 ||
                        regexec(&write_rx, fn, 0, NULL, 0) != 0)
                        allow = 0;
            }
            if (allow == 0)
                fprintf(stderr, "denying open: %s\n", fn);
        } else {
            fn[255] = '\0';
            fprintf(stderr, "str too long: %s\n", fn);
            return 0;
        }
    }
    return allow;
}
```

References

- [1] Linux security modules. <http://lsm.immunix.org/>.
- [2] Medusa DS9 security system. <http://medusa.formax.sk/>.
- [3] Rsbac. <http://www.rsbac.de/>.
- [4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2002.
- [5] M. Coleman. Subterfuge. <http://subterfuge.org/>.
- [6] Computer Emergency Response Team (CERT). Incident note IN-2003-03, Aug. 2003. http://www.cert.org/incident_notes/IN-2003-03.html.
- [7] Computer Emergency Response Team (CERT). Incident note IN-2004-01, Jan. 2004. http://www.cert.org/incident_notes/IN-2004-01.html.
- [8] G. Fernandez and L. Allen. Extending the UNIX protection model with access control lists, 1988.
- [9] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Network and Distributed System Security Symposium*, 2003.
- [10] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *USENIX*, 1996.
- [11] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.
- [12] T. Morlier. S4G: A sandbox for grids. <http://s4g.gforge.inria.fr/>.
- [13] N. Provos. Improving host security with system call policies. Technical Report 02-03, University of Michigan CITI, Nov. 2002.
- [14] W. R. Stevens. *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley, 2004.
- [15] R. N. M. Watson. Adding trusted operating system features to FreeBSD. In *USENIX Technical Conference*, June 2001.