


Sandboxing programs

G. Sébastien Chan-Tin

Department of Computer Science

Brown University

Submitted in partial fulfillment of the requirements for the Degree of Master of Science
in the Department of Computer Science at Brown University



Prof. Steven P. Reiss, Advisor

4-7-09
Date

Sandboxing programs

Sébastien Chan-Tin Steve Reiss
Computer Science Department
Brown University

Abstract

We have implemented a sandboxing tool for Linux that allows the user to control and isolate parts of the execution of a running process. Each isolated portion is given the illusion that it is interacting with a real system, while it is in fact confined to a virtual world. Our tool resides entirely in user-space and does not require changes to the operating system kernel or to the subject process. The sandbox has applications in the areas of security, debugging, transaction control and many others. To demonstrate its usefulness, we have integrated it with a debugging tool and with a testing framework where any changes made to files by test cases are discarded, making them idempotent and therefore repeatable.

1. Introduction

As part of the software development process, it is customary to write small programs to test the application being developed. These are given inputs with known correct outputs, and if the actual output matches the latter, then the program is deemed correct. If they do not match, then the application is debugged and the test is run again. However, these test programs often modify the state of the system, e.g. they write to files, and these changes must be undone before the tests can be run again, as they may be invalid otherwise. Undoing the changes can be a tedious task and prone to human error. Ideally, this process should be automated. We have developed a sandbox that can do that and more. Test cases become idempotent and can be repeated.

Broadly defined, a sandbox is a facility that creates an environment for a program to run. For example, the Java security framework is comprised of a sandbox within which untrusted code is forced to run. Access to resources outside the Java Virtual Machine (JVM) is monitored and controlled. Similarly, our sandbox allows users to control the execution of an arbitrary process (which we call the *subject program* or the *subject process*) and isolate it from the underlying system. The environment that our sandbox creates is a virtual one and changes made to it are discarded when the program terminates.

Sandboxes are related to virtual machines in that they both provide a virtual environment for processes to run. The main difference is that virtual machines provide a much larger virtual environment than do sandboxes: they emulate an entire system. Hence, virtual machines can only isolate an entire process from the underlying system. On the other hand, sandboxes only create a relatively smaller virtual environment and can isolate only a part of the subject process from the rest of the process and the underlying system. For example, our sandbox allows the user to isolate a thread. Fine grained isolation is important in multi-threaded applications such as web servers.

In this paper, we describe the implementation of our sandbox tool with two main characteristics. It allows users to control the execution of a process and isolate parts of it from each other. The various parts are isolated by including them in *contexts* of execution that will be kept separated from other contexts. Changes in one context are visible only in that context and not in others. Furthermore, a context can span multiple threads in the same process¹.

Essentially, a context is a *virtual world* in which part of the process executes. In order to isolate virtual worlds from each other, the sandbox buffers all interactions with the underlying physical system, working on copies of the real resource (files, etc) as necessary. In other words, a virtual world is a mapping of virtual resources onto real resources. When a virtual world is destroyed all of its local changes are lost, unless they are committed to the real world.

Such a tool can be useful in a number of situations. We describe two in more detail later in this paper, namely in debugging and in a testing framework. We give an overview of possible uses here.

¹ The tool could be extended so that contexts can span threads in different processes and even different machines.

As briefly described earlier, the sandbox could be used in a **testing framework** making test cases idempotent and hence allowing their repetition. Since the sandbox allows isolation of individual threads, multi-threaded applications such as web servers can dedicate a subset of their threads for the sole purpose of running test cases. This ability has interesting implications. For example, a web service provider could offer an interface for testing, where potential users can feed it some input and can inspect the output to determine whether it is as expected. Thus, this scheme offers a means to verify the semantics of third party applications. The key observation here is that there is no need for a separate server process and no need to restart the server.

The sandbox could also be used for **transaction management** by providing the means to checkpoint a process. In this case, the sandbox could be used to isolate transactions from each other. Such isolation comes naturally in the sandbox's environment when contexts are used to represent transactions. When a transaction is committed, the sandbox would apply the changes for the associated context, and when it is aborted, the sandbox would discard them. The testing framework can be thought of as a weaker form of transaction management.

A more common application for sandboxes is **security**, where they provide confinement capabilities. For example, Janus [8] and Alcatraz [14] provide an environment for running untrusted applications. In the former, restricted operations cause the program to abort whereas in the latter, changes to local files are temporary until the user accepts them. Similarly, our sandbox can provide confinement capabilities through a virtual environment within which to run untrusted applications. In this case, the entire program is run in one virtual world, and when the program exits, this virtual world is destroyed, causing all changes to be discarded. This situation is ideal if the program is suspected to contain malicious code capable of corrupting local files. With the sandbox, the user is still able to execute the program without fear of losing data. Additionally, the sandbox could also be used as part of a security framework, much like in the Java virtual machine, where operations that affect entities outside the context are validated against some security policy to determine whether they should be allowed or not. For finer grained validation, this could be combined with a data flow analysis approach that propagates the security attributes of data items from the time they enter the system, e.g. when they are read from disk or received from a remote entity on a network, to the time when they leave, e.g. when they are written to disk (possibly along with their security attributes).

With its ability to create virtual worlds, the sandbox lends itself as a **monitoring tool**. A virtual world is a simpler representation of reality. Hence, it becomes easier to monitor the subject program and then to deduce its behavior and semantics.

It also has applications in the area of **debugging**. A context provides the means to capture the partial state of a program, specifically the external modifications, e.g. data written to files or over a socket. Thus, a debugging tool could be built on top of the sandbox to take advantage of this facility. It could then compare the differences between two runs with different inputs to determine whether their outputs are different. With certain extensions, the sandbox could also be made to record state changes for the purposes of backtracking.

Our goal was to build a sandbox that can handle these various semantics in a generic manner and provide the hooks for specialization. Our sandbox alleviates the need for one-time development of specialized code for the required operations.

Another goal was to allow the use of the sandbox without modifications to the underlying system. We do not want to change the operating system itself because it might introduce instability and may make users reluctant to use our tool. We also do not wish to rely on specialized hardware which would make it too restrictive. Moreover, we do not want users to have to modify their programs. In certain situations, modification is not possible, such as when the source is not available or too difficult to modify and recompile. In others, it may not be desirable to modify the source code, for example, the latter could be a standard benchmark and changes should not be made to allow the experiments to be repeatable by others.

The rest of this paper is structured as follows. In section 2, we discuss related work. In section 3, we discuss the design and implementation of the sandbox. In section 4, we discuss results of using the sandbox to solve problems in two areas. Then, in section 5, we discuss our experiences and, in section 6, we discuss future work. We conclude in section 7.

2. Related Work

2.1. Sandboxing systems

A number of sandboxing systems have been proposed with goals similar to ours. In [12], Jones proposes a toolkit for interposition over system calls, with the objective being to allow users to specify the new semantics at a level higher than the system call themselves. Many of the suggested uses and goals of the system match ours. However, his implementation is in the form of a library which we cannot use, as we explain in the next section.

Similarly, in [11], Jain and Sekar developed a generic infrastructure for system call interposition on UNIX variant systems. Their infrastructure is partitioned in such a way as to allow the system-specific modules to be replaced for each platform. Their work is mostly geared towards abstracting away the complexities of process control from the user. Our work differs from theirs in that we allow finer grained control through the use of contexts.

Alcatraz [14] is a sandboxing tool that also buffers file I/O. It supports much of the semantics that we expect for our testing framework. However, it does not support the more complex semantics that we need for debugging, such as multiple contexts. Also, it is unclear how they handle multiple threads and sockets, both of which are prevalent in applications such as web servers.

In [8], Goldberg et al. describe a sandboxing tool built on the Solaris platform where helper applications can be confined. Their tool monitors the confined application and uses a policy definition to determine which resources it can access. If it attempts to access a denied resource, the application is aborted. This tool is not as sophisticated as ours and would not be easily extended to achieve the goals that we have listed above. MAPbox [1] is another sandbox implementation that focuses on ease of use by allowing users to define its behavior through templates representing classes of applications. Consh [3] is yet another sandbox implementation that focuses on transparent resource access across a network.

Some traps and pitfalls of system call interposition are given in [7]. It is based on Garfinkel's own experiences in building a similar system. We designed our system to take care of the applicable problems that are listed. Specifically: our system interposes on a more complete list of system calls and maintains more state information; we thoroughly tested our implementation²; we use absolute paths and traverse links before deciding whether to grant access to a resource; we copy file paths to a local buffer before invoking the real system calls to prevent another thread from changing it.

User-level sandboxing has also been suggested, in [18], as a means to extend the functionality of operating systems. Specifically, West and Gloudon suggest creating a shared virtual memory area that lies between user space and kernel space. Kernel extensions can be placed there and multiple processes can share this space. The implementation is more complex and requires modifications to the kernel and translation look-aside buffers for efficiency.

2.2. Isolation systems

A number of systems have tackled the issue of dividing up a process into multiple protection domains for isolation. In [17], Wahbe et al. propose an entirely software based system where code and data are divided into various segments, called fault domains, which are isolated from each other. A similar idea is used in the Java sandbox model [9] where Java classes are loaded with different permissions as defined in a security policy file. Each unique set of permissions makes up a protection domain and a class belongs to one and only one such domain. J-Kernel [10] takes a different approach by using capabilities as handles onto resources in other domains. These resources can only be accessed through capabilities. However, these approaches focus on cross-domain resource sharing with their main concern being security during such operations. In contrast, our focus is on complete isolation, such that portions of the same process are unaware of each other, and our targeted applications include more than just security.

² One of the listed problems was incorrectly mirroring OS code

3. Design and Implementation

3.1. Overview

The sandbox has two main tasks: control and isolation of contexts. The first is to trap and route all actions that operate on the external environment so that they instead operate on the virtual world for the current context. Essentially, the sandbox maps a virtual world onto the real world of the underlying operating system and hardware. This is done in two parts. First, the sandbox needs to find out when the process is about to interact with its environment. Then, it needs to change the interaction so that the process operates on the virtual world instead (which is really a mapping onto the real world).

The list of actions to control is not limited to ones that can modify the real world, but also ones that can read from the real world. For example, the subject process has to be able to read back what it wrote to disk. Thus, care must be taken to ensure that the sandbox has control over all necessary operations.

The second task is to separate changes in one virtual world from others. This is achieved by using context-specific data structures and a number of temporary files to hold the changes. Since changes can be large, it was decided not to store them in memory, but instead keep them on disk.

Our design is influenced heavily by our requirement that the sandbox be run without recompilation of the subject program. This means that the sandbox cannot simply be a library such that we can rewrite the subject program to use its functions. It also means that there can be only one context for unmodified applications. For programs where modifications are acceptable, an Application Programming Interface (API) in the C language is provided for integration with the sandbox. An API in the Java language is also provided, using the Java Native Interface (JNI) to interface with the C API. Thus, Java programs can use the sandbox without recompiling the Java Virtual Machine.

Early on, we experimented with making the sandbox as an interpose library, but it turned out not to be the right technique as we had to interpose on every C library function. We could not just interpose on a small subset of basic functions (e.g. `open`) since higher-level functions (e.g. `fopen`) were statically bound to the real versions of the basic functions.

In the end, we decided to use the same functionality available to debuggers, namely the `ptrace` system call, although implementing this is more complex than implementing a library. It allows one process to set itself up as a parent of another process and can thus modify or inspect the child. It can also setup breakpoints (a trap instruction) and be notified when the child process invokes a system call. In this sense, our sandbox is more similar to a debugger than anything else. It traps calls to standard C library functions (which make system calls to the underlying operating system) and modifies the instruction pointer (i.e. it simulates a jump instruction) to the appropriate sandbox function that can emulate this function. Thus, the sandbox keeps a mapping from standard C library function to sandbox function. Details of the trapping and mapping are given in the next section.

The sandbox is made up of two pieces: a driver piece and a library piece. The driver is responsible for setting up the subject process and receiving traps from it, while the library piece contains the sandbox functions that emulate the real system. Figure 1 gives a high-level architectural view of the sandbox tool. The driver's main responsibility is control of the subject process, while the library is responsible for isolation. The virtual worlds are managed by the library and their data reside in the same process space as the subject process. The sandbox library functions eventually call the standard C library functions, i.e. they do not re-implement these functions but merely wrap around them. The reader should note that having two distinct pieces forces us to create two initialization functions, one for each.

An additional complexity is that the sandbox needs to be aware of the various threads in the subject process. Each could be doing different things, such as making various library calls. Hence, the driver needs to maintain state information on a per-thread basis.

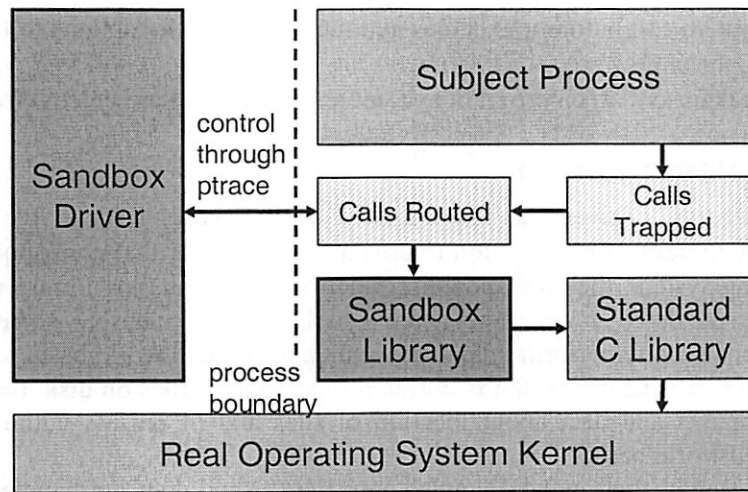


Figure 1: Architecture of the sandbox tool

3.2. Techniques

In this section we describe in more detail the techniques we use in our implementation.

The driver is responsible for starting the subject process and preparing it. There are two parts to this. First, the driver must setup the `ptrace` relationship to indicate to the operating system that it wants to control the subject. Second, it must load the sandbox library into the process space of the subject. Both require the driver to fork a new process and execute the subject program. Doing the first part is easy and is simply a specific call to `ptrace`. To do the second part, the driver sets the `LD_PRELOAD` environment variable to point to the library piece. This environment variable specifies any libraries that should be loaded in addition to any that were linked to it at compile time. Having the system load the library is a major gain as it saves us from pushing code from the driver into the process space of the subject – this is harder to implement correctly and is significantly slower.

The driver then needs to set up the mapping from the standard C library to the sandbox library. In order to be notified when a standard C library function is invoked, the driver sets up a breakpoint at the start (before any instruction is executed) of each appropriate C library function. We call such a breakpoint an *entry breakpoint*. When the trap is caught, the driver does a lookup to find out to which sandbox library function to route this call. Routing is done by setting the instruction pointer to the start of the sandbox library function. Care is taken to ensure that the latter takes the arguments of the same type and in the same order as the C library function. Thus, the sandbox library function can pop its arguments right off the stack.

In order to determine the addresses of both the C and sandbox library functions, we use the Executable and Load Format (ELF) [6] specification. The latter is a standard that defines how a process is laid out in memory and thus allows us to find the address of a function. In addition, we make use of the poorly documented `r_debug` structure, the rendezvous structure that contains information on where shared libraries are loaded. Combining the information obtained from reading the ELF structure of the subject program and reading the `r_debug` structure, we are able to obtain the start and end addresses of the necessary library functions.

When a thread of subject process reaches an entry breakpoint, it is said to enter *sandbox mode*. In this mode, any other entry breakpoints are ignored and simply stepped over. Before the thread is resumed, a breakpoint is set at the last instruction of the mapped-to sandbox library function. This breakpoint is called the *exit breakpoint* and when it executes, the thread exits sandbox mode. This mode is important as it helps the driver keep track of the state of each thread and prevents an infinite loop. Since the sandbox library functions call the C library functions, without tracking this mode, the driver will perpetually route these calls back to the calling sandbox library function.

In Linux 2.4, threads are really heavy weight processes which we also need to `ptrace`. By default, when a new thread is created, the `ptrace` settings are not carried over. This requires us to

intercept the `clone` system call before it executes and modify one of its options to force it to carry over the `ptrace` settings to the child.

Care was also taken so that, on program exit, all temporary files are deleted from disk.

3.3. Current implementation

The subject process is given the illusion of a virtual world through the use of virtual file descriptors. While the C library `open` function returns a file descriptor that corresponds to a resource on the underlying operating system, the corresponding function in the sandbox library returns a virtual file descriptor which corresponds to a virtual resource which, in turn, maps to some real resource. All sandbox library functions need to perform this mapping on a per-virtual world basis.

The actual data that is being written is buffered to temporary files on disk: the actual files are not modified. All the mappings and data about location of files and of buffers within files are stored in memory on a per-virtual world basis.

As it currently stands, the sandbox handles only file I/O and does not handle directories. It has a limited capability to handle sockets and pipes, and only sets them up so that there is a virtual correspondence to the physical socket/pipe. Minimal socket/pipe handling is critical as they share the same file descriptor table as files.

The sandbox also handles different contexts for different threads in the same process: threads are assigned a context and inherit their parent's context when created. It does not currently handle contexts that span multiple processes or machines. Also, the notion of a global context was implemented so that any file/socket/pipe opened in this context would be visible in all other contexts.

The sandbox supports two modes for a context: `OFF` and `SIMULATE`. In `OFF` mode, changes are applied directly to the files as if the sandbox were not there. `SIMULATE` mode is the mode where changes are simulated and disappear when the context is destroyed. The default mode can be specified when starting the driver.

So far, the sandbox has only been tested on Debian Linux running version 2.4 of the kernel. Due to the change in the implementation of threads, the sandbox will not work in Linux 2.6 as is. Thread handling will have to be updated in the driver.

4. Experimental Results

4.1. Elision

The sandbox has successfully been integrated with Renieris's Elision tool [15]. In this project, we want to determine whether taking a branch in an "if" statement has any effect on the output of the program. To achieve this, Elision uses the `fork`³ capability of Linux to duplicate a process's state. It modifies the resulting processes, such that the "mother" process takes the normal path and the "child" process takes the other. However, due to the fork, it is difficult for Elision to control the program in such a way as to ensure that all processes have the same input, and to capture and verify their outputs. The sandbox's role is to help in this respect. It buffers the input and output of the mother process, and controls the child processes so as to feed them the same input as the mother and to verify that their output is the same as the mother. Any mismatches are then reported to Elision.

We made some assumptions about subject programs in Elision in order to simplify the problem. We assumed that subject programs are monotonic, in the sense that they cannot undo any of their actions. For example, they only write to the end of files, not at random locations within them. Furthermore, we are interested only in "simple" programs that interact with the system only through files and the console, but not through a network. Also, we did not want to modify any of the subject programs, meaning that each contains exactly one context: the entire program. Lastly, we are only interested in differences in the visible output of the program, i.e. data written to files or to the console.

³ Only the mother process is forked to prevent an explosion in the number of states to explore.

4.1.1. Integration

Evidently, Elision requires changes to the sandbox that are specific to Elision in order to perform the necessary comparisons between the children and the mother. These changes were easily implemented. They involved adding one function to the list of captured library calls (`fork`), modifying three of the sandbox library functions (`open`, `read` and `write`) and modifying the sandbox library initialization and exit functions.

The `read` function was modified so that if the mother process reads from the console, the input is written to a file, which the child processes will read instead of the console. The `write` function has been modified so that, for the children, instead of writing, it instead reads back from the file and compares the buffer to be written and the buffer read from the file. We call this process *comparison writing*. If the two buffers differ, the child is terminated and a signal is issued to the mother process.

Changes to the `open` function involve adding special handling when opening a console file (`/dev/tty`). If it is opened for reading, a new file is created where the input data can be buffered. If it is opened for writing, a new file is instead created where this output will be written. Later, during comparison writing, this file will be read back.

Since the virtual world is stored in the same memory space as the subject process, a `fork` replicates the entire virtual world of the mother in the child. This is good for files opened in read mode, but not for files opened in write mode because we do not want to overwrite the output of the mother process, only compare with it. Hence, a `fork` has to close all files opened in write mode and reopen them in read mode. Similarly, files created by the mother process also have to be reopened in read mode. A `reopen` also involves setting the file pointer to the position of the last write. This operation and our assumption of monotonic programs guarantee that we read the appropriate buffer during comparison writing.

The semantics of `fork` have also been modified so that all children start in a blocked state. The mother process has to run to completion first because the mother process has to finish writing all of its data before a child can compare with them. When the mother process exits, it wakes up all children by issuing a signal.

In addition, on exit, the mother process waits for all of its children to terminate and gathers information on the termination status of each. It also ensures that the running time of each child does not exceed the running time of the original process multiplied by three. This handles the case when taking the other path of an “if” yields an infinite loop.

The sandbox library initialization function has also been changed to force buffering of the first three file descriptors (0 through 2). These represent the standard input (`stdin`), output (`stdout`) and error (`stderr`) descriptors. For `stdin`, a new file has to be created to store the input data. For both `stdout` and `stderr`, the output has to be redirected to a temporary buffer file.

A problem that we encountered and that is endemic to Elision is the fact that it creates a large number of processes, one for each “if” encountered at run-time. Due to operating system limits on the number of processes, we changed the `fork` function to only spawn a predefined bunch of children. On the next run (with an appropriate environment variable set) it will fork only the next bunch.

4.1.2. Other solutions

There are a number of other approaches that modify the behavior of the program, either statically or at run time. Dynamic mutation testing [13] establishes the sensitivity of code by changing function return values. Delta Debugging experiments with splicing parts of the state of a succeeding execution onto a failing one in order to isolate cause-effect chains [19]. Elision’s approach is more restrained: it only changes boolean values in conditions. This frees us from the need of an alternate run, and allows us to exhaust the space of changes. Critical slicing [5] selectively removes statements to examine whether they affect a slice. For conditionals, this technique would not always examine both branches. Mutation testing [3] modifies the subject program, and the user has to augment a test suite until it distinguishes all mutants from the original program. Elision can be used this way, where the user would have to build a test-suite in which all conditionals matter at least once.

Other debugging approaches manage the state of running programs. For example, in dynamic slicing and execution backtracking [1], the program is analyzed at runtime to find out which statements affect the value of a particular variable, then execution backtracking reverses these statements to return to a previous state. With this technique, one can backtrack to a previous state, reverse a conditional and resume. However, it lacks the ability to record the state at program termination and compare with the state when a conditional is reversed.

It may be possible to adapt sandboxes that record state, such as Alcatraz [14], for Elision. However, it is not clear how they would handle the specialized semantics for Elision, such as fork and buffering. Furthermore, our support for multiple contexts implies that our sandbox keeps track of changes within a file, while other sandboxes do not provide such support and only keep track of which files were changed. Thus, comparison writing is easier with our sandbox.

4.2. Testing in the Taiga environment

The sandbox is an important piece of the testing framework for Taiga, Reiss's "One World, One Program" project [15]. It was integrated without any modifications. In this framework, users provide a test case with known results to verify the implementation of a component. Without a sandbox, this test case could affect the component in a permanent way. However, if it is run within the sandbox, the latter can isolate the running of the test case from the rest of the application and any changes would disappear after the context is destroyed. Thus, with the sandbox, the test case is idempotent and can be run over and over with the same result. This is a weaker form of transaction management where the entire process is a transaction which is always aborted on exit.

An alternative solution to the sandbox would be to create a specialized testing framework. Since Taiga is written in Java and runs on a JVM, it could extend the facilities provided by the latter. Alternatively, the framework could be implemented from scratch in Java. However, this ties Taiga to the Java platform and would prevent it from being used on other platforms. For example, Taiga could be extended to include programs written for the .NET framework, but these programs would be unable to use the Java-specific testing framework and a new one would have to be written. Although the sandbox is tied to the Linux platform, it can serve a wider range of programs than if it were language-specific.

5. Experiences

Building the sandbox was an interesting experience and brought us to a number of rare situations. We outline some exotic problems that we ran into and provide a brief description of our solutions.

5.1. Stepping over a breakpoint in a thread-safe manner

The traditional solution to step over a breakpoint in debuggers is to put back the original instructions, single-step and then put back the trap instruction. However, this technique cannot be used in our case. It would cause a race condition on the breakpoint address as the other threads are still running. All threads share the same code in memory and unsetting the breakpoint for a thread could cause another to execute past it. Hence, we use a "scratch pad" where we copy the original instruction, single-step over it, and then jump to a point past the breakpoint. There is no race condition on the scratch pad because the sandbox driver is single-threaded and is the only entity that modifies it.

5.2. Deadlocks with one lock

It is not possible to have a deadlock with only one lock. However, we ran into synchronization constructs that are not locks. For example, we had two threads communicating over a pipe, with one thread writing and the other reading. Before doing either system call, we take the only lock in the sandbox. However, the read is a blocking operation. If the reader thread executes first, it will take the lock, read and block, while the writer thread will take the lock and block, creating a deadlock. The read operation will only wake up if it can read characters from the pipe, but this will never happen because the writer thread is blocked. The solution is to release the lock just before performing blocking calls. At this

point, there is no longer a contention for the virtual world's data structures. We had similar issues with send and receive pairs over sockets, and with connect and accept pairs for socket creation.

6. Future Work

One obvious area where the sandbox could be improved is its support for more platforms. In order to be more useful, the sandbox should be able to function on platforms other than Linux 2.4. For example, it should run on Linux 2.6, the latest version, so that it can continue to be useful.

Currently, the sandbox only supports a minimal form of transaction handling. Specifically, it is used as a testing framework, where changes are always discarded. A more complete transaction management system should be built to improve its usefulness.

The current trend in computing is moving towards distributed systems, i.e. systems made up of small components running on separate machines connected through a network. The World Wide Web is an example of such a system. To cope with this trend, the sandbox should be able to handle distributed contexts, i.e. contexts that are shareable across processes running on different machines connected to a network. Changes made by one of the processes should be visible to all processes that share the same context. Such a distributed sandbox would then be able to handle transactions distributed across these processes. It should also be able to more easily handle security in such an environment. A shared context makes it easier for the user to share authentication, access rights and other security-related information.

One of the limitations of the sandbox is that it can be circumvented if a programmer compiles his/her program against a static version of the standard C library. For such programs, the sandbox driver would be unable to find the addresses of the standard functions. A programmer could also make direct system calls by setting the appropriate registers and executing the appropriate interrupt. A better sandbox should instead trap system calls such as in [11].

7. Conclusion

We described the implementation of a sandbox for the Linux 2.4 platform using the ptrace functionality. It is able to capture system calls and route them to its own functions that emulate these calls. Thus, the sandbox creates and manages a virtual world with which a subject process interacts instead of the real world. The sandbox also supports multiple contexts, each with a virtual world that is isolated from all others. This allows the sandbox to separate the execution of various parts of the subject process.

The sandbox has a number of interesting uses and we described two in detail. It has been integrated in a debugging environment to find out whether "if" conditionals are elided. For each "if", the subject process is instrumented so that it forks, with the parent process taking the normal path and the child process taking the other. Virtual worlds are used to buffer input to and output from the subject program in order to determine whether the output of all the copies are exactly the same. Our tool is also used in a testing framework, where test cases are run, but any modifications that they make are not committed. This is a weaker form of transaction management. The greatest advantage in this environment is that test cases become idempotent, making them repeatable. In both cases, the extensions required of the sandbox were small.

8. References

- [1] Anurag Acharya and Mandar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Applications. In *Proceedings of the USENIX Security Symposium*, 2000.
- [2] Hiralal Agrawal, Richard A. DeMillo and Eugene H. Spafford. Debugging with Dynamic Slicing and Backtracking. *Software-Practice and Experience*, 1993.
- [3] A. Alexandrov, P. Kmiec, and K. Schausser. Consh: A confined execution environment for internet computations. Available at <http://www.cs.ucsb.edu/~berto/papers/99-usenix-consh.ps>, 1998.
- [4] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: help for the practicing programmer. *IEEE Computer*, April 1978.

- [5] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *Proceedings of the international symposium on Software testing and analysis*, January 1996.
- [6] Executable and Linkable Format (ELF). Tool Interface Standards, *Portable Formats Specification V1.1*.
- [7] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition based Security Tools. In *Proceedings of the Network and Distributed System Security Symposium*, February 2003.
- [8] Ian Goldberg, David Wagner, Randi Thomas and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the USENIX Security Symposium*, July 1996
- [9] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *Proceeding of the USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [10] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu and Thorsten von Eicken. Implementing Multiple Protection Domains in Java. In *Proceedings of the USENIX Annual Technical Conference*, June 1998.
- [11] K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *Proceedings of the Network and Distributed System Security Symposium*, February 2000.
- [12] Michael B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the Symposium on Operating Systems Principles*, December 1993.
- [13] Janusz Laski, Wojciech Szermer, and Piotr Luczycki. Dynamic mutation testing in integrated regression analysis. In *Proceedings of the international conference on Software Engineering*, 1997.
- [14] Zhenkai Liang, V. N. Venkatakrishnan and R. Sekar. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In *Proceedings of the Annual Computer Security Applications Conference*, December 2003.
- [15] Steven P. Reiss. Personal communication. spr@cs.brown.edu, 2002-2004.
- [16] Manos Renieris, Sébastien Chan-Tin and Steven P. Reiss. Elided Conditionals. To appear for publication at the workshop on *Program Analysis for Software Tools and Engineering*, 2004.
- [17] Robert Wahbe, Steven Lucco, Thomas E. Anderson and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Symposium on Operating Systems Principles*, December 1993.
- [18] Richard West and Jason Gloudon. User-Level Sandboxing: a Safe and Efficient Mechanism for Extensibility. Technical Report, 2003-014, Boston University, June 2003.
- [19] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2002.