Retaining Sandbox Containment Despite Bugs in Privileged Memory-Safe Code

Justin Cappos, Armon Dadgar, Jeff Rasley, Justin Samuel, Ivan Beschastnikh,
Cosmin Barsan, Arvind Krishnamurthy, Thomas Anderson

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
{justinc,armond,jeffra45,jsamuel,ivan,cosminb,arvind,tom}@cs.washington.edu

Abstract

Flaws in the standard libraries of secure sandboxes represent a major security threat to billions of devices worldwide. The standard libraries are hard to secure because they frequently need to perform low-level operations that are forbidden in untrusted application code. Existing designs have a single, large trusted computing base that contains security checks at the boundaries between trusted and untrusted code. Unfortunately, flaws in the standard library often allow an attacker to escape the security protections of the sandbox.

In this work, we construct a Python-based sandbox that has a small, security-isolated kernel. Using a mechanism called a security layer, we migrate privileged functionality into memory-safe code on top of the sandbox kernel while retaining isolation. For example, significant portions of module import, file I/O, serialization, and network communication routines can be provided in security layers. By moving these routines out of the kernel, we prevent attackers from leveraging bugs in these routines to evade sandbox containment. We demonstrate the effectiveness of our approach by studying past bugs in Java's standard libraries and show that most of these bugs would likely be contained in our sandbox.

Categories and Subject Descriptors

C.20 [**General**]: Security; D.4.6 [**Security and Protection**]: Security kernels

General Terms

Security, Languages

Keywords

Sandbox, Layering, Containment

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

1. INTRODUCTION

Programming language sandboxes, such as Java, Silverlight, JavaScript, and Flash, are ubiquitous. Such sandboxes have gained widespread adoption with web browsers, within which they are used for untrusted code execution, to safely host plug-ins, and to control application behavior on closed platforms such as mobile phones. Despite the fact that program containment is their primary goal, flaws in these sandboxes represent a major risk to computer security [23].

A sandbox is divided into three components: the core language interpreter, the standard libraries, and the untrusted application code. The standard libraries contain routines to perform file I/O, network communication, cryptography, math, serialization, and other common functionality. These libraries need to contain some native code, usually written in C or C++, but many sandboxes follow security best practices [33] and implement the bulk of their standard libraries in a memory-safe language like Java or C#. While flaws in native code pose an obvious risk [35, 54], many flaws in memory-safe code are also a threat [18, 12, 46, 47].

For example, in Java 1.4.2, the serialization methods of the Calendar object had a security bug which was caused by instantiating an object of a broader type than intended [12, 32]. This mistake allowed an attacker to deserialize a new security policy and use it to permanently elevate all privileges. Despite the flaw residing entirely in memory-safe code, this allowed a malicious party to escape the sandbox and perform malicious actions, such as installing malware. This is not an isolated incident; many major security vulnerabilities in Java have been found in Java code instead of native code [42].

In this work, we mitigate the impact of a bug in memory-safe privileged standard libraries by isolating libraries from each other and from the interpreter. Instead of relying on a single monolithic trusted computing base (TCB), we construct a small, self-contained sandbox kernel and place sensitive portions of the standard libraries in isolated components on top of it. For example, in our system the serialization library runs in an isolated component that has the same permissions as application code. Flaws in this library will not help an attacker escape the sandbox.

Specifically, we construct standard libraries as a set of isolated and contained *security layers*. Each security layer is untrusted by its ancestor security layers, but is trusted by its descendant security layers. The lowest security layer

(with no ancestors) is called the sandbox kernel and is the only portion of the libraries where a flaw may allow an attacker to escape the sandbox. At a high level, security layers are similar to well-tested layering techniques used in prior systems [19, 31]. This technique, however, has not generally been used in widely used web-based execution environments. In the development of our sandbox, we created a language called Repy, which is a subset of Python version 2.5.4. Repy provides isolation restrictions which are similar to an object-capability language. In addition to the usual restrictions imposed by an object-capability language, the boundary between two security layers is monitored by an *encasement library*, which verifies interface semantics at runtime. This verification prevents capability leaks and minimizes the risk of time-of-check-to-time-of-use (TOCTTOU) bugs. As a result, security layers provide similar protection to separate processes that communicate using remote procedure calls (RPC). However, the performance of a security layer call is significantly better than a local RPC invocation.

The contributions of this work are as follows:

- We design and implement an appropriate set of abstractions for constructing strong, yet flexible security layers.
- We identify functionality that can be migrated out of the kernel into higher security layers, thus reducing the security impact of a bug in the sandbox code.
- We describe how a security layer is a natural mechanism for adding customized security policies that are *transparent* to an application.
- We evaluate the security and performance implications of using security layers and discuss limitations and optimizations.

The remainder of this paper is organized as follows. Section 2 illustrates flaws and limitations of the standard library implementations in existing sandboxes in more detail. In Section 3, we discuss the goals of our system and give a brief overview of the architecture. Section 4 discusses the construction of security layer functionality. Following this, Section 5 describes how the functionality in standard libraries is divided between the sandbox kernel and security layers. In Section 6, we present different applications for security layers based on our experience with a live deployment. We evaluate the performance of our security layer implementation in Section 7. Relevant prior work is discussed in Section 8, and Section 9 concludes. Appendix A details the Repy language that underlies this work.

2. BACKGROUND

In this section, we describe our threat model and examine the popular Java sandbox execution environment in detail. Our sandbox environment is constructed on top of Python and thus is not directly comparable to Java. However, it is useful to motivate the construction of a new sandbox by examining the techniques other sandboxes use to isolate privileged standard library code. As Java's primary implementation is open source, we examine it in detail. Further, we also found that Silverlight, and its open source implementation Moonlight, possess similar features including a single, large TCB. Our work does not address faults in JavaScript, Adobe Flash, or Google Native Client since their standard library implementations are not memory-safe.

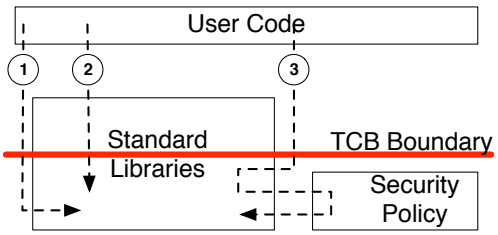

Figure 1: Java's architecture, especially the standard libraries and their relationship to its single TCB. There is no clearly defined boundary between the privileged and untrusted portions of the standard libraries. Dotted lines indicate possible call-paths from user code into the TCB. From left to right: (1) user code may directly call privileged library code in the TCB; (2) user code may call unprivileged library code which may then call into the TCB; (3) user calls may be subject to security policy, as determined by the invoked standard library code in the TCB.

2.1 Threat Model

A process has a set of privileges provided by the operating system, usually exposed as system calls. The primary security goal of a sandbox is to restrict a program to some subset of privileges, usually by exposing a set of functions that mediate access to the underlying operating system privileges. The attacker's goal is to obtain access to privileges that were not intentionally exposed by the sandbox, thus escaping the sandbox.

To attack a sandbox, we assume an attacker may run multiple programs in different sandboxes on the same machine. In addition, the attacker may use multiple threads to modify visible state or issue concurrent requests in an attempt to trigger a race condition. An attacker may submit arbitrary code for execution and may pass data of any type the interpreter allows to any visible calls. Once the code begins executing, the attacker's program can manipulate any object in its namespace in any way that the interpreter allows.

Given the large amount of code in the standard library for each sandbox, we assume that an attacker may have knowledge of flaws in this code. Our goal is to prevent bugs in this code from allowing an attacker to escape the sandbox.

2.2 Learning from Java

Java developers implemented various security mechanisms to allow the Java Virtual Machine (JVM) to be run as a secure sandbox. For instance, Java code cannot perform unsafe operations, such as modifying arbitrary memory locations, due to restrictions placed on it by the Bytecode Verifier [41] and the JVM. Java programs can, if allowed, call directly into native C code, which may perform unsafe operations, such as call system calls and modify arbitrary memory locations, on their behalf. To provide isolation, Java does not grant untrusted code with unmediated access to native C code. Instead, the sandboxed code is typically allowed to call some subset of the pre-existing native code that is part of the standard libraries. Sensitive native functions are often scoped to be less than public, and access to them is mediated by public Java wrapper functions. Such

wrapper functions will verify access and enforce a security policy for its standard libraries, using a variety of security components, such as the ClassLoader, SecurityManager, and AccessController. Figure 1 overviews Java's architecture.

However, Java's TCB is much larger than native code and the security policy. All of the Java code which mediates access to sensitive native calls is also clearly a security risk. However, the visibility of sensitive native calls extends much further than functions that wrap calls. Java's standard libraries are organized into packages that the programmer may choose to import. The code is organized by functionality type, rather than privilege, so privileged code is contained across many different packages. Objects may contain a mixture of privileged and unprivileged methods and data members. The scope of an object's data members, however, must extend to at least the containing file (and sometimes extends to the entire package). This grants untrusted code direct access to sensitive native functions. To get an idea of the scope of native code intermingling, we examined all Java objects in Java 1.6.18 and found that there are about 1800 native method calls spread around 500 objects. Out of the Java objects that have at least one native call, approximately 350 restrict scope to at least one native method by setting the visibility to private. This means that large portions of Java's standard libraries may perform actions that should be restricted for security reasons.

To summarize, Java's security model is such that a very large amount of Java code must be correct in order to maintain security. This extends to complex components like the ClassLoader, AccessController, and SecurityManager. Based upon the scope of sensitive native code, the TCB also extends to portions well beyond just the wrapping functions in the standard libraries. Experience has shown that if any of these pieces has a security flaw, all of the protections of the sandbox may be compromised [48, 42]. Flaws in the standard libraries of Java pose a significant risk to 4.5 billion devices worldwide [30] despite considerable security focus from both industry and academia.

3. GOALS AND OVERVIEW

The goal of this work is to ensure that a security failure in the standard library code has minimal security impact. To achieve this, our sandbox provides the vast majority of its functionality in memory-safe library code where faults will not result in an attacker escaping the sandbox. Of course, it is unavoidable for our sandbox to have at least some privileged code to allow access to the operating system, but it is possible to minimize the quantity and complexity of this code. To realize this goal, we want to construct and organize a set of libraries such that:

- The risk of compromise is minimized by moving library functionality out of the kernel to the maximum practical extent.
- Custom security policy functionality exists entirely outside of the kernel.
- It is trivial for a developer to prevent common bugs such as capability leaks and race conditions.
- With our changes, libraries remain easy to develop and to use.

As a basic building block for isolation, we constructed a custom subset of Python called Repy that is similar to an object-capability language (described in Appendix A). Our sandbox is comprised of a small, trusted kernel, a set of required libraries, standard libraries, and user code as shown in Figure 2. Each security layer obtains a set of capabilities when it is instantiated by the security layer beneath it. A security layer is isolated and may only interact through the security-verified set of capabilities it is provided. A vulnerability in a security layer can at most allow the compromise of the security layers it instantiated. Through such a compromise the attacker cannot gain any capabilities that are stronger than those already granted to the compromised security layer. Since the sandbox kernel maintains containment over its descendants, only a vulnerability in the sandbox kernel may lead to escape of the sandbox.

Our design allows much of the functionality that languages usually provide to be executed with the same capabilities as the untrusted user code. As we will describe in Section 5, we can build significant portions of functionality usually found in the TCB, such as module import, in untrusted user code.

Over the past 20 months, we have used the described sandbox as part of a network research testbed [49]. The testbed is built using donated resources from end user machines, in a manner similar to BOINC [11] and SETI@Home [3]. However, since this is a network testbed, the use model is closer to PlanetLab [43]. Typical use cases of our testbed include network measurement, peer-to-peer applications, web measurement, and distributed hash tables.

As the size of our testbed grew into the thousands, machine owners began requesting increasingly more complex functionality. There were requests to control the local IP or interface used, the source or destination of network traffic, different resource restrictions based on the system location, and other functionality. This motivated us to add required security layers as a general mechanism to help facilitate these requests. Unlike in Java, the standard library programmer does not need to remember to add security policy checks in the appropriate portions of the standard libraries. In our model, a security layer that wishes to enforce a policy, such as the same origin policy enforced by web browsers, may simply interpose on the network capabilities that are permitted to the security layer it instantiates. As we discuss in Section 6, this is a common way to interpose required security functionality without bloating the TCB or breaking compatibility with existing code.

4. SECURITY LAYER DESIGN

This section describes how the sandbox provides security layer functionality within our architecture. First, we describe *virtual namespaces*, which are provided by the kernel for loading and executing code. We then describe the *encasement library*, which is implemented above the TCB and uses virtual namespaces to implement the security layer abstraction.

4.1 Virtual Namespace

To support code loading and execution, the sandbox kernel supports two virtual namespace calls. The first call validates the safety of code. It takes a string that contains the program source and ensures that the string only contains language constructs permitted by our Repy language (see Appendix A for more details).

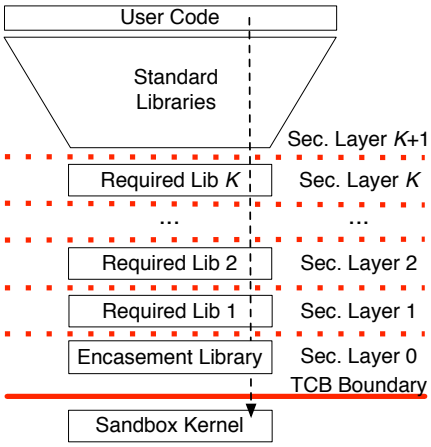

Figure 2: Architecture of our sandbox with a minimal sandbox kernel comprising the TCB. Each required library is isolated in its own security layer (indicated with horizontal dashed lines), and standard or optional libraries are located in the same security layer as user code (top layer). There is a clearly defined boundary between each security layer. The bottommost security layer contains the encasement library, which enforces security layer isolation. The vertical dotted line indicates the only possible call-path from user code into the sandbox TCB.

The second virtual namespace call executes validated code with the provided capability mapping. The namespace will not contain capabilities from the sandbox kernel or the namespace of the creating module unless these are explicitly listed in the provided capability mapping. For example, if module `foo` with capabilities `listfiles`, `safe_removefile`, and `removefile` were to instantiate a module `bar` with a capability mapping `{'listfiles':listfiles, 'removefile':safe_removefile}`, the module `bar` would have access to `foo.listfiles` via the name `listfiles` and to `foo.safe_removefile` via the name `removefile`. The module would be unable to access `foo.removefile`.

4.2 Encasement Library

The virtual namespace abstraction is useful for loading code dynamically, but does not provide adequate security for use as an isolation boundary. A created virtual namespace will share objects with the creator and may be missing even basic verification such as type checking.

The encasement library implements the security layer abstraction, which provides strong isolation between virtual namespaces. The call to create a new security layer takes three arguments: the code to isolate in a security layer, a capability mapping, and a *contract* that specifies capability semantics. Security layers do not share objects or functions. The encasement library copies all objects that are passed between security layers.

Each function call that can be called by other security layers is wrapped in a verification function. The verification function uses the contract for a function to verify its behavior. A contract specifies the set of behaviors that a verification function should have. The verification function and contract are conceptually similar to system call filtering mechanisms [24, 9, 28, 1], in that they mediate access to a sensitive interface. For example, since Python is a dynamically typed language, it is useful to type check a function's arguments, exceptions, and return values. The contract lists the number and types of arguments, the exceptions which can be legally raised by the function, and the return type of the function.

A contract is represented as a Python dictionary which is a hash table with keys and values. As an example, if the module `foo` wanted to create a contract that would map `listfiles` and `safe_removefile` into a new namespace, the contract would be: `{'listfiles':{'type':'func', 'args':None, 'exceptions':None, 'return':list, 'target':listfiles}, 'removefile':{'type':'func', 'args':str, 'exceptions': (RepyArgumentError, FileNotFoundError, FileInUseError), 'return': None, 'target':safe_removefile}}`. Note that the symbols listed in the contract come from `foo`'s namespace. Thus the target for the `'listfiles'` in the contract is the `foo.listfiles` capability. Similarly, the target for the `'removefile'` in the contract is the `foo.safe_removefile` capability.

The verification function uses the contract to perform type-checking whenever a capability is used. If the verification function detects a semantic violation, the program is terminated. In addition to type checking, the verification function copies arguments and return values of mutable types to prevent time-of-check-to-time-of-use bugs. Since mutable types are copied, the caller cannot cause a race condition by modifying objects.

4.3 Security Layer Instantiation

Each security layer call provides the encasement library with a contract to instantiate the next security layer. Eventually the final security layer starts the user program with the appropriate set of capabilities. This instantiation process is helpful when a developer wants to implement a security layer that interposes on a specific capability. This is done by substituting a version of the function that enforces a given policy. All security layers loaded after this layer will have access to the version of the function which enforces this new policy. Since every layer above the interposition layer has access only to the new version of the function, the user's program is forced to use the new policy.

From start to finish, the entire process proceeds as follows. The sandbox kernel obtains a list of command-line arguments, the first of which must be the encasement library. The kernel reads in the encasement library code and uses the virtual namespace abstraction to execute the code with the exported kernel capabilities[1]. The encasement library uses its security layer creation call to instantiate the next security layer. To do this, the encasement library creates a capability mapping that contains the kernel's exported capabilities, the security layer instantiation call, and the remaining command-line arguments. The newly instantiated security layer repeats this process using the encasement library's security layer creation call to instantiate the next security layer with a potentially updated contract and capability mapping. Eventually, the user program is instantiated in a separate security layer with the capabilities provided through the stack of security layers that preceded it.

[1]The kernel wraps its calls similarly to how the encasement library works.

5. STANDARD LIBRARIES

The security layer mechanism effectively isolates the sandbox kernel from the functionality that can be externalized. However, it is important to retain containment while externalizing functionality. For example, if the ability to write to arbitrary files on the file system is allowed externally to the kernel, this could be used to overwrite the sandbox implementation's code. While there is insufficient space to fully describe the capabilities provided by our sandbox kernel implementation [25], we summarize its capabilities and then describe how our sandbox provides functionality common to existing sandboxes.

Our sandbox kernel has a total of 32 capabilities that it provides to the untrusted security layer above it. These calls can be summarized as follows:

- Thirteen network functions, to perform DNS lookups, obtain the local IP address, and send / receive TCP and UDP traffic.
- Two virtual namespace calls described in the previous section.
- Six file I/O calls involving access to a sandbox-specific directory on the file system. These allow the user to open a file, read at a location in the file, write at a location in the file, close the file, delete a file, and list the files in the sandbox.
- A call to create and return a lock object, which has methods to acquire or release the lock.
- Four functions to provide information. These calls return a string to describe the last error's stack trace, resource utilization information, the thread's name, and the elapsed time since the program started.
- Three thread-related calls: a call to create a new thread of execution for a function, a call to sleep the current thread, and a call to force all threads to exit.
- A call to return random bytes suitable for cryptographic use.

Using the above primitives, we have built standard libraries that reconstruct common language functionality. For example, `import` is provided via a library that calls the sandbox kernel to read the appropriate file from disk, and passes the string containing the code into the kernel's virtual namespace API. All of the complexity of correctly mapping symbols between namespaces is handled outside of the sandbox kernel. Utilizing the minimal functionality provided by the sandbox kernel we were able to restore access to large amounts of Python functionality including `print`, `eval`, traceback handling, and many types of introspection.

Similarly, standard libraries can extend the minimal file system API provided by the sandbox kernel to provide many conveniences expected by programmers. For example, in Python a programmer can iterate over the lines of a file using `for line in file:`. Rather than providing an iterator for thread-safe readline with consistent file location in the kernel, we provide it as part of a standard library above the TCB. Similarly, we provide write-buffering, logging functionality, and other common mechanisms in standard libraries, moving thousands of lines of code out of the TCB.

While reconstructing the common language functionality of Python we worked with numerous undergraduates to implement our standard libraries. These undergraduates were able to write a majority of our standard libraries within roughly five to ten hours per library with a heavy focus on testing. For certain library functions there existed a pure-Python implementation already; in this case we found that occasionally the function ran correctly within our sandbox without any major modifications.

In addition to core functionality, such as import and basic file I/O, other complex functionality is also provided outside of the sandbox kernel. This includes cryptographic libraries, XML parsing, RPC, serialization, NAT traversal, HTTP client / server code, argument parsing, advanced synchronization primitives, and a variety of encoding schemes. These routines comprise the majority of the lines of code in our codebase. A bug in any of these routines will not allow an attacker to escape the sandbox. Somewhat surprisingly, portions of these libraries are implemented with security sensitive code in other sandbox environments.

Another benefit of moving complexity out of the kernel is that it simplifies interposition. For example, the kernel function for opening a TCP connection requires that ports and IP addresses are explicitly specified for both sides of the connection. If a security layer wants to prevent connections from using specific local IPs, or to reject traffic to specific destinations, such filtering policies are trivial to implement because all connection-related information is explicit. This explicitness is also useful to expand the minimal API for programmer convenience. For instance, programmers may want to be able to specify the remote side of the connection as a hostname instead of as an IP address. We provide this functionality in a portion of the standard library that is loaded after all the other security layers.

In the next section we overview a particular feature of our sandbox architecture – required libraries. These libraries are loaded in security layers prior to any standard libraries or user code. We overview how we leverage required libraries to implement security policies and to carry out a variety of useful functionality, all of which is enabled by the interposition afforded by security layers.

6. REQUIRED LIBRARIES

Previously, our sandbox allowed the machine owner to filter application capabilities using mechanisms such as regular expressions over the values allowed as arguments to privileged calls. This complex set of functionality resided entirely in the trusted computing base.

The interposition provided by security layers made our previous approach obsolete – it is trivial to perform filtering using Python code in a security layer. This allowed us to reduce the size of the sandbox kernel by removing regular expressions and the other filtering code. Using security layers also naturally separates the policies specified by the machine owner from policies that are controlled by the application developer. As Figure 2 shows, the application developer can only load security layers once the *required libraries* – security layers required by the sandbox and the machine owner – have been loaded. In this section we describe an assortment of interesting required libraries that we have implemented in our sandbox.

6.1 Network API Interposition for Controlled Node Communication

A networked testbed must be designed with a global security perspective in mind. For example, the testbed should not be able to send SPAM, launch DDoS attacks, spread malware through remote exploits, or perform similar malicious actions. To prevent such actions, there must be restrictions on which hosts testbed nodes can communicate with. Specifically, it is desirable to allow testbed nodes to only communicate with each other, with critical services like DNS, and with computers that have explicitly opted-in. To provide this capability, we constructed a traffic containment service [7], which restricts communication between the local node and a remote node based on the destination IP, port, or some combination of the two. Here we briefly overview how required libraries made this service possible.

Although this service is essential to the adoption of our testbed, it is also complex due to the various distributed protocols it uses to maintain the list of participating nodes. Because of this we were hesitant to add this service to the TCB. Instead, we isolate this service in a security layer, and transparently add it as a required library to those machines whose owners want to filter the source or destination of their machine's traffic. The transparency of security layers makes the addition of this required library trivial – it merely interposes on the sandbox network API just above the kernel. Additionally, security layer isolation guarantees that a bug in this library will not compromise the sandbox kernel.

6.2 Transparent Forensic Logging

To help investigate potential abuses, to support auditing, and to help debug our platform we wanted to collect various local information, such as the destination of traffic and the rate of resource consumption. An important requirement of this collection is that the researcher using the machine should not be able to modify the collected data.

We implemented this functionality in a required library by interposing on the calls that need to be logged and then writing out the collected data into a file. To prevent modification or deletion of this file, the library traps the `openfile` and `removefile` capabilities. Leveraging security layers enabled us to move this functionality outside of the sandbox kernel, without sacrificing transparency for the application code.

6.3 Dynamic Policy Loading

An administrator may want to add or remove security layers for a large number of machines under their control. However, making the change on each system manually is time prohibitive. We constructed an *administrator control* required library, which retrieves a list of security layers to load from a user-specified server. This list is signed with the administrator's private key (along with other security metadata to prevent replay attacks), and contains the names, locations, and secure hashes of the security layers to be loaded.

6.4 Location-Aware Resource Restriction

Required libraries also make it easy to construct adaptive policies. An example of this is a policy that changes depending on the machine's location. We implemented such a policy as a required library that periodically checks the machine's IP address and changes the capabilities set based on the machine's geolocation. This allows a user to have different network restrictions when their laptop is at home or at work, for example.

7. EVALUATION

In this section we evaluate our approach. First we compare the resiliency of our sandbox to the JVM. We do so by considering previous security bug reports for the Java code portion of the JVM [42]. We manually translate these bugs into the context of our sandbox to understand how they would manifest. We categorize the impact of these bugs on our sandbox and detail how security layers help to mitigate them. Second, we evaluate the cost of our techniques by quantifying the performance impact and memory overhead of constructing and using security layers in Python.

7.1 Risk Reduction

To evaluate the change in risk with using security layers, we considered the set of critical Java security vulnerabilities studied by Paul and Evans [42]. For each bug, we attempted to understand how the vulnerability impacts our sandbox if the component containing the vulnerability were translated into our sandbox. Our translation was guided by considering how the buggy component would be implemented in our system, relying on the underlying motivation of our project to migrate as much functionality as possible out of the sandbox kernel. Because this evaluation effort is inherently qualitative we mitigated the subjective nature of the analysis by employing three authors to independently categorize the severity of each bug. The authors then actively discussed the bugs where they had any category disagreement until a complete consensus on the appropriate category for each bug was reached.

Table 1 describes the bugs, their severity in Java, and their categorized severity in our sandbox. The values in the security layer severity column of this table mean the following: Prevented – bug cannot manifest in our sandbox; Insufficient detail – bug report did not provide enough information to make the translation possible; Cannot translate – the bug is specific to the JVM and cannot be mapped into the context of our sandbox; Exception – an exception is raised when the bug is triggered; Hang – the bug hangs the sandbox; Allowed – the bug is explicitly allowed to manifest due to security policies of our sandbox kernel; Unknown – the bug does not have a single definitive translation.

The most critical bugs in Table 1 allow arbitrary code execution or read access to the entire file system. We now discuss these critical bugs in more detail in the order they appear in the table.

- Leveraging the bug in CVE-2001-1008 an attacker can execute signed code with expired signatures. This is a risk because this code may be native code containing flaws. Our sandbox prohibits users from executing native code (whether signed or not). The closest translation of this bug would manifest as a security layer that only loads code if the code is signed. A similar flaw in this layer would allow malicious code to be loaded and executed, however the malicious code would not be able to escape the sandbox.

- CVE-2005-3905 and CVE-2005-3906 provide insufficient information to make a translation possible. If the reflection vulnerabilities exist due to bugs in type

Bug Number	Short Description	Java Severity	Security Layer Severity
CVE-2001-1008	Execute apps with expired signatures	Arbitrary	Prevented
CVE-2005-3905/3906	Reflection vulnerability 1 & 2	Arbitrary	Insufficient detail
CVE-2002-0865/0866	XML/DB vulnerability allows code loading	Arbitrary	Prevented
CVE-2002-1293	CAB loader missing security	Arbitrary	Prevented
CVE-1999-0766/0141	Bytecode verifier buffer flaw 1 & 2	Arbitrary	Cannot translate
CVE-1999-0440	Bytecode verifier loads an unchecked class	Arbitrary	Prevented
CVE-2000-0327/2002-0076	Bytecode verifier cast bug 1 & 2	Arbitrary	Cannot translate
CVE-2003-0111	Bytecode verifier unknown bug	Arbitrary	Cannot translate
CVE-2004-2627	Bytecode verifier bytecode bug	Arbitrary	Cannot translate
CVE-2003-0896	A '/' in a classname bypasses security	Arbitrary	Prevented
CVE-2000-0676	Can use `file:///` to read arbitrary files	File Read	Prevented
CVE-2000-0162	Remote attacker can read files	File Read	Prevented
CVE-2000-0711	Incorrect `ServerSocket` creation	Socket	Allowed
CVE-2000-0563	Arbitrary connection via HTTP redirection	Socket	Allowed
CVE-1999-0142/1262	Arbitrary outgoing connections 1 & 2	Socket	Allowed
CVE-2002-0058	Session hijack if client uses proxy	Socket	Allowed
CVE-2002-0867	Handle validation crashes JVM	Unclear	Exception
CVE-2002-1289	No address validation for native services	Unclear	Prevented
CVE-2003-0525	Double free from `getCanonicalPath`	Unclear	Prevented
CVE-2002-1287	Long classname crashes JVM	Crash	Prevented
CVE-2005-3583	Deserialization may crash JVM	Crash	Prevented
CVE-2004-1503	Integer overflow in DNS raises exception	Exception	Exception
CVE-2004-2540	`readObject` may hang the JVM	Hang	Hang
CVE-2004-0651	Unknown vulnerability	Hang	Insufficient detail
CVE-2002-1292	Can manipulate class loading security	Unclear	Prevented
CVE-2004-0723	Cross-site file system communication	Information leak	Prevented
CVE-2002-1260	DB access missing checks	DB access	Unknown
CVE-2002-1290	Applets can manipulate clipboard	Clipboard access	Unknown
CVE-2002-1288/1325	Applet can discover PWD 1 & 2	Directory Info	Unknown
CVE-2002-0979	Applet may write executable to known path	Unknown	Unknown

Table 1: A listing and description of JVM security vulnerabilities from [42] that we translated into the context of our sandbox to study the risk reduction benefits of our sandbox. For those bugs where the translation was feasible, the last column indicates the severity of the bug once it was translated. Values in this column are defined in the text. The bugs are sorted according to the order in which they are discussed in the text.

conversion or memory safety, similar bugs would allow arbitrary code execution in our sandbox as well. However, if the flaws are related to incorrect capabilities when using reflection, there is no security risk as this portion of our implementation (e.g. `eval`) exists in a security layer.

- CVE-2002-0865, CVE-2002-0866, and CVE-2002-1293 are the result of different standard libraries using custom code loading mechanisms. In our sandbox, custom code loading functionality would be implemented inside the security layer hosting the standard library (e.g. XML, CAB). This would prevent the vulnerability.
- CVE-1999-0766, CVE-1999-0141 CVE-2000-0327, CVE-2002-0076, CVE-2003-0111, and CVE-2004-2627 cannot be easily translated because the bug descriptions have inadequate information. These bugs are in the bytecode verifier which is significantly different in Java and Python. Our implementation does not load Python bytecode, but instead passes the interpreter the program's source (Appendix A). However, our implementation would prevent CVE-1999-0440 (a missing check for bytecode security), since the only way to load code is through a single call provided by the sandbox kernel.
- The critical vulnerability CVE-2003-0896 and file access bugs like CVE-2000-0676 and CVE-2000-0162 exist because of insufficient checks on file system access. In our implementation, there is a single set of routines in the sandbox kernel that all routines must use. We prevent this category of flaws by placing the appropriate file system access check in a single place.

One category of bugs listed in Table 1 deals with violations of the same-origin policy (CVE-2000-0711, CVE-2000-0563, CVE-1999-0142, CVE-1999-1262, and CVE-2002-0058). As our sandbox is in part used for networking research, our network policy is more permissive than the same-origin policy. Namely, our users typically choose between allowing arbitrary network connections (no security layer interposition) or use a security layer like the Controlled Node Communication layer (Section 6.1).

There is also a large class of bugs that crash or hang the JVM (CVE-2002-0867, CVE-2002-1289, CVE-2003-0525, CVE-2002-1287, CVE-2005-3583, CVE-2004-1503, CVE-2004-2540, and CVE-2004-0651), and possibly lead to more serious attacks. In our framework, these bugs most likely result in exceptions or hang the sandbox, thus having a similar denial-of-service effect.

The security policy manipulation bug (CVE-2002-1292) exists because Java had outdated security checks that an attacker can manipulate. Since valid security policies rarely use this interface, the main impact of the vulnerability would be to allow a malicious party to prevent the loading of arbitrary classes. A similar vulnerability would exist in our

	No args	Immutable	Mutable	Exception
General	5.9 μs	12 μs	15 μs	8.8 μs
Custom	.15 μs	.80 μs	1.7 μs	1.1 μs

Table 2: Call overhead for general and customized security layer verification routines.

framework if an outdated security layer with a vulnerability was loaded on client machines.

The final classification of bugs are those that leak sensitive information or access, but do not allow escape of the sandbox (CVE-2004-0723, CVE-2002-1260, CVE-2002-1290, CVE-2002-1288, CVE-2002-0979, and CVE-2002-1325). It is difficult to directly translate these issues to our sandbox. However, in general using a small sandbox kernel makes boundaries and capabilities explicit, which may mitigate the confusion that led to some of these errors.

7.2 Performance

In many applications there exists a delicate balance between performance and security. In the context of our application – hosting experimental code on volunteered machines – we can reduce performance to increase security. In this section we evaluate the two types of performance penalties incurred when using security layers: initialization and use. All of the following performance tests were run on an Apple iMac with an Intel i7-860 2.8GHz CPU, 4GB of 1333MHz RAM running Mac OS X 10.6.3. Running time calculations are averages over 10,000 iterations.

Initialization Cost. When a security layer is initialized, the code is validated and the encasement library creates custom wrapper functions for the individual functions. To evaluate this, we examined the initialization time of the sandbox with and without the encasement library. The sandbox itself takes about 135 ms to initialize and the encasement library takes another 38 ms. In addition, each security layer takes time to initialize, as the dispatch method must wrap the functions and objects necessary to maintain containment between each security layer. We also found it takes 2.5 ms to initialize a security layer that adds a noop function and then dispatches the next security layer. We believe this overhead is acceptable given that Java (1.6.0_20) and Python (2.6.1) start in 170 ms and 17 ms respectively.

Per-use Cost. The second type of performance cost is incurred whenever a security layer is crossed[2]. Table 2 shows the cost for our general encasement library implementation, which inspects and validates the contract of a capability at run time, and a customized version which performs the same operations but is optimized ahead of time. In each case, the type of the arguments or exceptions that the function raises has a slight performance impact, but the costs stay roughly within the same order of magnitude. This means that the type of contract does not significantly impact the performance of the encasement library for arguments and exceptions of a small size.

To evaluate overall performance, Table 2 shows that the cost of a general contract verification implementation is approximately an order of magnitude higher than that of a customized version. However, the general contract verification implementation is much easier to verify for correctness and security properties than a customized version (hence we use the general verification in production). The performance penalty is only paid when crossing the boundary between two security layers. Each security layer that interposes on a call typically performs functionality that is much more expensive than this. To put these numbers in perspective, a function call is about an order of magnitude cheaper than a customized security layer crossing. However, in Python a function call performs no type checking or other validation so the native mechanisms are clearly inadequate to provide security isolation. The most appropriate security comparison is with a local RPC, which provides the same sort of security guarantees as a security layer. A local RPC using XML-RPC is three orders of magnitude more expensive than crossing the security layer boundary using our general contract verification implementation.

[2]Note that a security layer which does not interpose on a call, does not impact its running time.

7.3 Memory Overhead

In addition to consuming additional CPU when using multiple security layers, each security layer also incurs a memory cost. This cost is due to the space needed by the contracts, the copying of mutable arguments, and the wrappers needed for security layer crossings. In our experiments on a system with 64-bit memory addresses, each security layer consumes about 19 KB of memory. The encasement library consumes an additional 1 MB of memory. Our experience has been that these memory overheads are acceptable, even on memory-limited devices such as mobile phones.

8. RELATED WORK

We gained significant inspiration from previous ideas and systems. At the highest level, our security layer abstraction is conceptually similar to layering in prior systems [19, 31]. We categorize and discuss closely related work below.

8.1 Object-Capability Based Languages and Systems

Capabilities in computer systems have a long history [34]. In particular there has been a significant amount of work on object-capability languages [39, 38], which provide a principled set of techniques for resolving the long studied problem of dividing an application into security contained components [51, 29, 16]. Joe-E [38] is the only other object-capability language that uses a subset of a widely used programming language (Java). Current work on Joe-E is focused on application security and although the authors are cognizant that the lack of security in the standard library code of the Joe-E sandbox is a limitation of their technique, it remains an open problem that has not been addressed [38]. In this work, we address this limitation using the security layer primitive to isolate trusted code. Work by Stiegler and Miller [53] on a capability subset of OCaml demonstrated that object-capabilities do not have to impact the language's expressivity or performance. Their work has similar limitations to Joe-E – privileged standard libraries are excluded and all authorities from the safe version of the standard library must be removed.

Our system has many conceptual similarities to the Hydra capability-based operating system [34] in that we provide similar guarantees and mechanisms for protection, albeit for a programming language sandbox instead of an operating system. One significant difference is that our protection mechanism (encasement library) is not inside of our sand-

box kernel. Thus a flaw in the encasement library can at most allow one to bypass security layers but will not allow escape of the sandbox. Another difference is that we do not allow a process to do a 'rights walk' through the set of local namespaces and utilize capabilities. This allows calls and objects to pass both ways through security layers without compromising security, which is important for callbacks and notifications.

Cannon, et. al. [13] present a method for securing the Python interpreter by implementing a set of resource restrictions. They modified the Python interpreter to prevent read/write/execution of arbitrary files/modules. Using this modified interpreter they added the ability to import external modules, based on a user defined whitelist. This leaves the sandbox integrity up to the user, which can be problematic. If a user whitelists a module, such as 'sys', it would give the interpreter file I/O capabilities which could allow the modification of the whitelist. In our implementation a user is unable to add functionality to a security layer that could produce undesired side effects such as giving full access to the file system.

8.2 Secure Language Subsets and Isolation

Prior work by Back, et. al. [4] encourages a single, explicit separation between trusted and untrusted code, which they call "the red line." Our motivation is similar, but we propose a separation between multiple security layers, with a small kernel of truly trusted functionality.

Other work has gone into building secure language subsets that restrict the allowed operations and functionality in a language [37, 22, 58]. There are various subsets of JavaScript, such as Facebook's FBJS [22], Yahoo's ADsafe [58] and formal attempts from within the language [37]. While this is useful for isolating an untrusted program's namespace, it does not allow security functionality to be composed in a manner similar to security layers.

Isolation techniques have been applied to other domains such as extensions for web browsers [8], operating system separation of processes [2], virtual machine separation of operating systems [27, 6], constraining the functionality of a process [36], or running mutually distrustful programs within a single process [56, 17, 5]. In this work, we focus on the converse problem – given a mechanism for isolation between components, we leverage secure interposition and interaction to construct secure standard libraries.

8.3 Interposition

In addition to interposition by language restrictions, some researchers have also used conceptually similar mechanisms to interpose on a process, usually at the system call layer [45, 44]. Current OS interposition mechanisms have several drawbacks that make them undesirable in practice. For example, they are OS specific, may require the user to install kernel patches, require the interposition code to be trusted, incur significant overhead, and are prone to subtle errors [57, 26]. Our use of security layers is a lightweight means of achieving similar functionality, while performing interposition by a straightforward wrapping of function calls.

Another mechanism for interposition is to rewrite the user program to contain references to the appropriate monitor code [21, 55, 14]. Rewriting was used by Erlingsson, in his work on Inlined Reference Monitors [21] and has also been used in conjunction with aspect-oriented programming [55, 14]. We could have used similar techniques to add security checks, but decided against performing modification of source code because of the difficulties in asserting the correctness of these techniques. We believe that using separate namespaces makes it much easier for a programmer to reason about the behavior and correctness of a security layer.

8.4 System Call Filtering

There has also been a significant amount of work on system call filtering as a mechanism to secure existing system call interfaces [24, 28, 1, 9]. While the low-level mechanisms are similar to our approach, prior work focuses on securing an existing system call interface, while we leverage our flexibility to define the interface. This led us to make different design decisions.

In [24], Fraser, et. al. describe a technique for securing commercial off-the-shelf software with the use of generic wrappers. Their implementation uses a form of tagging to categorize system calls that a user-defined wrapper should be applied to. One type of tag that is used is a parameter tag, which is similar to our use of contracts. Parameter tags may be used to generate wrapper functions that perform argument copying (including deep copies of structures). One important difference is that they rely solely on static analysis of these tags which may limit the flexibility of the system.

Systems like Janus [28], MAPbox [1] and Tron [9] apply a wrapper function to system call contracts. These wrappers indicate acceptance or denial of calls. In our system, instead of restricting calls only via contracts, the resulting security layer actually executes code and so may directly return or raise an exception.

8.5 Distributed Virtual Machine

Gün Sirer, et. al. [52] propose a distributed virtual machine (DVM) architecture and use it to decompose the JVM into a set of system services, which are offloaded from the client machine. This reduces verification overhead, improves security through physical isolation, and allows a system administrator to verify code and enforce a security policy across all machines that they administer. Our use of security layers has similar benefits without requiring a centralized component. In our model, similar functionality is provided by dynamic policy loading (see Section 6.3).

8.6 Information Flow Control

There has been a lot of work on information flow control both in operating systems [20, 59] and programming languages [40, 50, 10, 15]. These techniques are used to tag and track data as it moves through an application. This work has features like isolation and control of interfaces that is also used in our work. Information flow techniques are complimentary but orthogonal to our use of security layers. Security layers focus on the capabilities and call semantics across security boundaries instead of tracking data flow. The most related information flow control work is Wedge [10]. Wedge is a system to modularize application logic so that it does not leak sensitive user information. Wedge focuses primarily on memory tagging, although it does utilize SELinux policies [36] to limit the set of allowed system calls. Wedge's notion of callgates could be used as a building block for system call interposition, but requires kernel support and at least one kernel crossing per call. Security layers do not have Wedge's memory tagging functionality, but security layers

are more lightweight than Wedge, perform boundary checking, and require no kernel support.

9. CONCLUSION

In this work we designed, implemented, and evaluated security layers – a mechanism to isolate and transparently interpose on standard libraries in a programming language sandbox. Security layers make it possible to push library functionality out of the sandbox kernel, thereby helping to mitigate the impact of bugs in libraries. As a result, our Python-based sandbox maintains containment of application code despite bugs in the standard library implementation. To evaluate our design we examined a set of known JVM security bugs and found that security layers would likely prevent at least 6 of the 8 applicable bugs that led to arbitrary code execution. Security layers also help to protect against vulnerabilities that led to arbitrary file reads, sandbox crashes, and other faults.

Our experience with a 20 month sandbox deployment across thousands of nodes has been overwhelmingly positive. Security layers allowed users to add security functionality, without increasing the risk of sandbox escape. We have used security layers to enforce network communication policies, log forensic information, and perform other operations, without adding any code to the sandbox kernel. In addition, when optimized for performance, security layers incur a performance penalty that is within an order of magnitude of a function call. Given the security and functionality benefits of security layers, we feel that this mechanism incurs an acceptable performance penalty and is broadly applicable, meriting consideration in the design of any sandbox.

Acknowledgments

We would like to thank the large number of people who helped to significantly improve this paper. We appreciate the feedback and discussions with Tadayoshi Kohno, Wenjun Hu, Mark Miller, Marc Stiegler, Adam Barth, and Adrian Mettler. We are also grateful to our shepherd Trent Jaeger and the anonymous reviewers for their valuable feedback.

This material is based upon work supported by the National Science Foundation under CNS-0834243. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of BBN Technologies, Corp., the GENI Project Office, or the National Science Foundation.

10. REFERENCES

[1] A. Acharya and M. Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2000. USENIX Association.

[2] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, page 10. ACM, 2006.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[4] G. Back and W. Hsieh. Drawing the red line in Java. In *HotOS'99*, pages 116–121, 1999.

[5] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: isolation, resource management, and sharing in Java. In *OSDI'00*, pages 23–23, Berkeley, CA, USA, 2000. USENIX Association.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP'03*, pages 164–177, New York, NY, USA, 2003. ACM.

[7] C. Barsan and J. Cappos. ContainmentInSeattle – Seattle – Trac. `https://seattle.cs.washington.edu/wiki/ContainmentInSeattle`. Accessed April 3, 2010.

[8] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. Technical Report UCB/EECS-2009-185, EECS Department, University of California, Berkeley, Dec 2009.

[9] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *In Proceedings of the USENIX 1995 Technical Conference*, pages 165–175, 1995.

[10] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: splitting applications into reduced-privilege compartments. In *NSDI'08*, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[11] BOINC. `http://boinc.berkeley.edu/`. Accessed April 2, 2010.

[12] A security vulnerability in the Java Runtime Environment (JRE) related to deserializing calendar objects may allow privileges to be escalated. `http://sunsolve.sun.com/search/document.do?assetkey=1-26-244991-1`. Accessed April 8, 2010.

[13] B. Cannon and E. Wohlstadter. Controlling Access to Resources Within The Python Interpreter. `http://www.cs.ubc.ca/~drifty/papers/python_security.pdf`. Accessed July 19, 2010.

[14] B. Cannon and E. Wohlstadter. Enforcing security for desktop clients using authority aspects. In *AOSD'09*, pages 255–266, New York, NY, USA, 2009. ACM.

[15] S. Chong, K. Vikram, A. Myers, et al. SIF: Enforcing confidentiality and integrity in web applications. In *Proc. 16th USENIX Security*, 2007.

[16] G. Czajkowski. Application isolation in the Java Virtual Machine. In *OOPSLA'00*, pages 354–366, New York, NY, USA, 2000. ACM.

[17] G. Czajkowski and L. Daynés. Multitasking without comprimise: a virtual machine evolution. In *OOPSLA'01*, pages 125–138, New York, NY, USA, 2001. ACM.

[18] D. Dean, E. Felten, and D. Wallach. Java security: From HotJava to Netscape and beyond. In *1996 IEEE Symposium on Security and Privacy.*, pages 190–200, 1996.

[19] E. W. Dijkstra. The structure of the "THE"-multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968.

[20] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP'05*, page 30. ACM, 2005.

[21] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell, 2004.

[22] FBJS - Facebook developers wiki. `http://wiki.developers.facebook.com/index.php/FBJS`. Accessed April 2, 2010.

[23] Pwn2own 2010: interview with charlie miller. `http://www.oneitsecurity.it/01/03/2010/interview-with-charlie-miller-pwn2own/`. Accessed July 26, 2010.

[24] T. Fraser, L. Badger, and M. Feldman. Hardening cots software with generic software wrappers. *Foundations of Intrusion Tolerant Systems*, 0:399–413, 2003.

[25] FutureRepyAPI – Seattle. `https://seattle.cs.washington.edu/wiki/FutureRepyAPI`. Accessed April 15, 2010.

[26] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS'03*. Citeseer, 2003.

[27] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP'03*, pages 193–206, New York, NY, USA, 2003. ACM.

[28] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the Wily Hacker. In *SSYM'96: Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, Berkeley, CA, USA, 1996. USENIX Association.

[29] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *USENIX ATC'98*, pages 22–22, Berkeley, CA, USA, 1998. USENIX Association.

[30] Learn about Java technology. `http://www.java.com/en/about/`, Accessed April 8, 2010.

[31] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A Retrospective on the VAX VMM Security Kernel. *IEEE Trans. Softw. Eng.*, 17(11):1147–1165, 1991.

[32] S. Koivu. Calendar bug. `http://slightlyrandombrokenthoughts.blogspot.com/2008/12/calendar-bug.html`. Accessed April 8, 2010.

[33] B. Lampson. Computer security in the real world. *Computer*, 37:37–46.

[34] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[35] S. Li and G. Tan. Finding bugs in exceptional situations of JNI programs. In *CCS'09*, pages 442–452, New York, NY, USA, 2009. ACM.

[36] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX ATC'01*, pages 29–40, 2001.

[37] S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. *Programming Languages and Systems*, pages 307–325.

[38] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed Systems Symposium*. Internet Society, 2010.

[39] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[40] A. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release at `http://www.cs.cornell.edu/jif`. Accessed April 3, 2010.

[41] S. Oaks. *Java Security*. O'Reilly and Associates, Inc., Sebastopol, CA, USA, 2001.

[42] N. Paul and D. Evans. Comparing Java and .NET security: Lessons learned and missed. *Computers and Security*, pages 338–350. Volume 25, Issue 5, July 2006.

[43] PlanetLab. `http://www.planet-lab.org`. Accessed April 2, 2010.

[44] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, volume 1, page 10. Washington, DC, 2003.

[45] PTrace. `http://en.wikipedia.org/wiki/Ptrace`. Accessed April 2, 2010.

[46] Fujitsu Java Runtime Environment reflection API vulnerability. `http://jvndb.jvn.jp/en/contents/2005/JVNDB-2005-000705.html`, Accessed April 8, 2010.

[47] Sun Java Runtime Environment reflection API privilege elevation vulnerabilities. `http://www.kb.cert.org/vuls/id/974188`, Accessed April 8, 2010.

[48] Section 5 – the three parts of the default sandbox. `http://www.securingjava.com/chapter-two/chapter-two-5.html`. Accessed April 8, 2010.

[49] Seattle: Open peer-to-peer computing. `http://seattle.cs.washington.edu/`. Accessed April 3, 2010.

[50] V. Simonet and I. Rocquencourt. Flow Caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, pages 152–165. Citeseer, 2003.

[51] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. In *EuroSys'06*, pages 161–174, New York, NY, USA, 2006. ACM.

[52] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *SOSP'99*, pages 202–216, New York, NY, USA, 1999. ACM.

[53] M. Stiegler and M. Miller. How Emily tamed the Caml. Technical Report HPL-2006-116, Advanced Architecture Program. HP Laboratories Palo Alto, 2006.

[54] G. Tan and J. Croft. An empirical security study of the native code in the JDK. In *Proceedings of the USENIX Security Symposium*, pages 365–377, Berkeley, CA, USA, 2008. USENIX Association.

[55] J. Viega, J. Bloch, and P. Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2):31–39, 2001.

[56] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *SOSP'94*, page 216. ACM, 1994.

[57] R. N. M. Watson. Exploiting concurrency

vulnerabilities in system call wrappers. In *WOOT'07*, pages 1–8, Berkeley, CA, USA, 2007. USENIX Association.

[58] Making JavaScript safe for advertising. `http://www.adsafe.org/`. Accessed April 2, 2010.

[59] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI'06*, pages 263–278.

APPENDIX

A. THE REPY LANGUAGE

This section describes how we restricted the Python programming language to a subset we call Repy, that is similar to an object-capability system. Python is a memory-safe language, in that, it does not allow a program to manipulate pointers or inspect memory directly. However, memory-safety is not sufficient for the construction of an object-capability system. While a similar effort has been performed for Java [38], Python has some interesting differences because the language allows much more introspection than Java.

A.1 Python Scopes

In Python, there are three scopes: *local*, *global*, and *built-in*. The local scope is associated with a function or a block of code, the global scope is associated with a module, and the built-in scope is always available. In Python each of these scopes is represented by a dictionary. The built-in scope contains the primitives, basic data structures, and common exceptions. Normally, this dictionary also contains unsafe functions such as import, and eval but we remove these during sandbox initialization. Our technique for isolating system code from user code is to define sensitive code in a separate global context and to control all references between the user and system code. The only permitted references from the user's global scope to the system's global scope are API calls, all of which are explicitly added to the scope before any user code executes. Since the local scope is created and destroyed at a function level, there is no need to handle it separately as it does not have access to anything not already available in the global scope.

A.2 Code Safety Evaluation

The goal of code safety evaluation is to test if the provided program can execute on the underlying runtime system without presenting a security risk. For this, a candidate program is validated by checking whether it meets a behavioral specification, which restricts the source code to just the valid operations. This analysis is performed on the interpreter's parse tree of a program and disallows unsafe instructions, such as the *import* instruction.

A.3 Programming Language

The sandbox virtual machine executes code written in a subset of the Python language. To minimize the risk of bugs, the virtual machine implementation attempts to build on parts of the underlying trusted computing base that are stable, conceptually simple and widely used. For example, we allow use of a vanilla type of the Python interpreter's style of classes and simple types; we do not allow classes that subclass basic types, provide their own namespace storage mechanisms, or utilize other complex mechanisms that are rarely used or new to the language.

This language supports a subset of Python language primitives and constructs – it is, in fact, executed by the Python interpreter. The virtual machine loads code as text and then uses the standard compiler module built into the interpreter to build a parse tree of the code. The virtual machine verifies that the parse tree contains only the supported subset of the language using a popular safety module, which was initially developed by Phil Hassey and that we significantly adapted to our context. If there is a disallowed language construct, the code is rejected *without executing*. We found this method to be efficient and robust in detecting disallowed or unrecognized functions and language constructs. Assuming the parse tree for the code contains the valid subset of Python, the code is executed once the built-ins and the API are mapped into its namespace.

The virtual machine verifies multiple aspects of the Python interpreter before it begins executing loaded code. Allowed built-in Python functions are checked to ensure their signatures (i.e. argument list) are defined as expected. This prevents differences in interpreter implementations or interpreter versions, which may result in changes to built-in functions, from exposing unintended functionality to untrusted code.

A.4 Language Built-ins

Static analysis of a parse tree will not catch all security threats. Many dynamic languages have built-in functions and object attributes that allow a high degree of introspection (referred to as reflection in Java), enabling code to inspect and manipulate elements of the execution environment. Introspection is useful in debugging, profiling, and (as we have found) in restricting the functionality of executing code. However, introspection also provides mechanisms for circumventing the API and language restrictions. To combat this, the virtual machine remaps the introspective built-ins to make them trap out of the sandbox and cause termination. This remapping essentially augments the interpreter to perform simple run-time analysis for further safety.

However, while language constructs constrain what loaded code executes, there remain certain built-in functions that provide functionality outside of what can be expressed within our language constraints. These built-ins must also be correctly restricted because execution of these unsafe functions may cause the code to escape the sandbox resource or isolation restrictions.

The set of allowed built-ins consists of 87 items obtained directly from Python's interpreter. These include definitions of constants and exceptions like True and ValueError (53), type conversion functions like float and chr (15), math functions like max and pow (8), as well as miscellaneous Python operations like len and range (11). Notice that operations like import (which includes code from another module), eval, and exec are not directly allowed because verifying their safety is known to be difficult. The virtual namespace primitive (Section 4) provides unprivileged code with access to equivalent functionality that is safely implemented.