Evaluating the Flexibility of the Java Sandbox

Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine
Carnegie Mellon University
{zfc,mmaass}@cs.cmu.edu, tding@cmu.edu, {clegoues,sunshine}@cs.cmu.edu

ABSTRACT

The ubiquitously-installed Java Runtime Environment (JRE) provides a complex, flexible set of mechanisms that support the execution of untrusted code inside a secure sandbox. However, many recent exploits have successfully escaped the sandbox, allowing attackers to infect numerous Java hosts. We hypothesize that the Java security model affords developers more flexibility than they need or use in practice, and thus its complexity compromises security without improving practical functionality. We describe an empirical study of the ways benign open-source Java applications use and interact with the Java security manager. We found that developers regularly misunderstand or misuse Java security mechanisms, that benign programs do not use all of the vast flexibility afforded by the Java security model, and that there are clear differences between the ways benign and exploit programs interact with the security manager. We validate these results by deriving two restrictions on application behavior that restrict (1) security manager modifications and (2) privilege escalation. We demonstrate that enforcing these rules at runtime stop a representative proportion of modern Java 7 exploits without breaking backwards compatibility with benign applications. These practical rules should be enforced in the JRE to fortify the Java sandbox.

1. INTRODUCTION

There are three broad reasons that Java is such a popular target for attackers. First, the Java Runtime Environment (JRE) is widely installed on user endpoints. Second, the JRE can and often does execute external code, in the form of applets and Java Web Start (JWS) applications [1, 2]. Finally, there are hundreds of known and zero-day vulnerabilities [3] in Java. In the common scenario, often referred to as a "drive-by download," attackers lure users to a website that contains a hidden applet to exploit JRE vulnerabilities.

In theory, such attacks should not be so common: Java provides a sandbox to enable the safe execution of untrusted code and to isolate components from one another. This should protect both the host application and machine from malicious behavior. In practice, these security mechanisms are problematically buggy such that Java malware is often able to alter the sandbox's settings [4] to override security mechanisms. Such exploits take advantage of defects in either the JRE itself or the application's sandbox configuration to disable the security manager, the component of the sandbox responsible for enforcing the security policy [5, 6, 7, 8].

In this paper, we investigate this disconnect between theory and practice. We hypothesize that it results primarily from unnecessary complexity and flexibility in the design and engineering of Java's security mechanisms. For example, applications are allowed to change the security manager at runtime, whereas static-only configuration of the manager would be more secure. The JRE also provides a number of security permissions that are so powerful that a sandbox that enforces any of them cannot be secure. We hypothesize that benign applications do not need all of this power, and that they interact with the security manager in ways that are measurably different from exploits. If true, these differences can be leveraged to improve the overall security of Java applications, and prevent future attacks.

To validate these insights, we conducted an empirical study to answer the question: How do benign applications interact with the Java security manager? We studied and characterized those interactions in 36 open-source Java projects that use the security manager, taken from the Qualitas Corpus [9] and GitHub.

We discovered two types of security managers in practice. *Defenseless* managers enforce a policy that allows sandboxed code to modify sandbox settings. Such applications are inherently insecure, because externally-loaded malicious code can modify or disable the security manager. We found defenseless managers in use by applications that modified sandbox settings at runtime, typically as workarounds to using more complicated (but more correct) security mechanisms or to enforce policies or implement functionality unrelated to security. We believe that such applications use the sandbox to implement certain non-security requirements because Java does not provide better mechanisms for doing so. The sandbox is not intended to be used this way, and these use cases both reduce security in practice and limit the potential exploit mitigations that are backwards compatible with benign applications. On the other hand, applications with *self-protecting* managers do not allow sandboxed code to modify security settings. It might still be possible to exploit such applications due to defects in the JRE code that enforces security policies, but not due to poorly-deployed local security settings.

ACSAC '15, December 07 - 11, 2015, Los Angeles, CA, USA

ISBN 978-1-4503-3682-6/15/12...$15.00

DOI: http://dx.doi.org/10.1145/2818000.2818003

We found that software that uses the sandbox as intended—for protection from malicious external code—does not use its vast flexibility. For these applications, the flexibility decreases their security without obviously benefiting the developers or application functionality. In fact, we found and reported a security vulnerability related to the sandbox in one of the applications under study.

We propose two runtime rules that restrict the flexibility of the sandbox and fortify Java against the two most common modern attack types without breaking backwards compatibility in practice. We evaluate our rules with respect to their ability to guard against ten applets in a popular exploit development and delivery framework, Metasploit 4.10.0[1], that successfully attack unpatched versions of Java 7. Taken together, the rules stopped all ten exploits and did not break backwards-compatibility when tested against a corpus of benign applications. We are engaged in an ongoing discussion on the OpenJDK security-dev mailing list about implementing runtime enforcement of these rules in the JVM itself.

The contributions of this papers are as follows:

- A study of open-source applications' interactions with the security manager (Section 4). We identify open-source applications that enforce constraints on sub-components via the Java sandbox, as well as unconventional behaviors that indicate usability and security problems that the Java security model can be improved to mitigate.
- An enumeration of Java permissions that make security policies difficult to enforce (Section 2.2), a discussion of real-world cases where these permissions are used (Sections 4.3 and 4.4), and a sandbox-bypassing exploit for a popular open-source application made vulnerable due to their use (Section 4.4).
- Two novel rules for distinguishing between benign and malicious Java programs, validated empirically (Section 5).
- A discussion of tactics for practically implementing the rules, with a case for direct JVM adoption (Section 5.1).

We begin by discussing necessary background on the Java sandbox and exploits (Section 2). We present the methodology and dataset for our empirical study in Section 3. The results of the study are discussed in Section 4, leading to our rules which are defined and evaluated in Section 5. Finally, we discuss limitations, cover related work, and conclude in Sections 6, 7, and 8 respectively.

2. BACKGROUND

In this section, we describe the Java sandbox (Section 2.1), distinguish between defenseless and self-protecting security managers (Section 2.2) and provide a high-level description of how Java exploits commonly work (Section 2.3).

2.1 The Java sandbox

The Java sandbox is designed to safely execute code from untrusted sources using components summarized in Figure 1. When a *class loader* loads a class (e.g., from the network, filesystem, etc.), it assigns the class a *code source* that indicates the code origin, and associates it with a *protection domain*. Protection domains segment the classes into groups by *permission set*. These sets contain permissions that explicitly allow actions with security implications, such as writing to the filesystem, accessing the network, etc [10]. Unlisted

[1]http://www.metasploit.com/

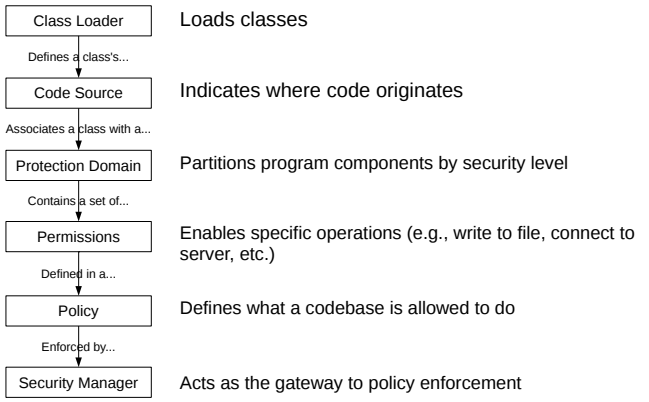

Figure 1: High-level summary of the Java sandbox.

actions are disallowed. *Policies* written in the Java policy language [11] define permission sets and their associated code sources. By default, all classes *not* loaded from the local file system are run within a restrictive sandbox that restricts their ability to interact with the host application or machine.

The sandbox is activated by setting a *security manager*, which acts as the gateway between the sandbox and the rest of the application. Whenever a sandboxed class attempts to execute a method with security implications, that method queries the security manager to determine if the operation should be permitted. To perform a permission check, the security manager walks the call stack to ensure each class in the current stack frame has the specific permission needed to perform the action.[2]

Missing checks in code that *should* be protected are a common source of Java vulnerabilities, because the security-critical code must initiate the check. Note that such vulnerabilities lie in the JRE itself (i.e., the code written by the Java developers), not in code using the sandbox to execute untrusted code.

2.2 Defenseless vs. self-protecting managers

Java is flexible about when the sandbox is enabled, configured, and reconfigured. The default case for web applets and applications that use Java Web Start is to set what we call a *self-protecting* security manager before loading the network application. The security manager, and thus the sandbox, is self-protecting in the sense that it does not allow the application to change sandbox settings. Self-protecting managers might still be exploited due to defects in the JRE code that enforces security policies, but not due to poorly-deployed local security settings. We contrast self-protecting managers with those we call *defenseless*, meaning that sandboxed applications are permitted to modify or disable the security manager. A defenseless manager is virtually useless in terms of improving the security of either a constrained application or its host. However, we find in Section 4 that developers have found interesting non-security uses for defenseless managers in otherwise benign software.

We evaluated whether each of the available Java permissions can lead to sandbox bypasses. Table 1 summarizes the set of permissions that distinguish between self-protecting

[2]Stack-based access control is discussed in more detail in [12, 13, 14, 15]

Table 1: List of sandbox-defeating permissions. A security manager that enforces a policy containing any of these permission results in a defenseless sandbox. A subset of these permissions were first identified in [7].

Permission	Risk
RuntimePermission("createClassLoader")	Load classes into any protection domain
RuntimePermission("accessClassInPackage.sun")	Access powerful restricted-access internal classes
RuntimePermission("setSecurityManager")	Change the application's current security manager
ReflectPermission("suppressAccessChecks")	Allow access to all class fields and methods as if they are public
FilePermission("<<ALL FILES>>", "write, execute")	Write to or execute any file
SecurityPermission("setPolicy")	Modify the application's permissions at will
SecurityPermission("setProperty.package.access")	Make privileged internal classes accessible

and defenseless security managers.

2.3 Exploiting Java code

While the Java sandbox *should* prevent malicious applets from executing their payloads, certain defects in the JRE implementation of these security mechanisms can permit malicious code to set a security manager to `null`.[3] This disables the sandbox and enables all operations. This approach was common in drive-by downloads between 2011 and 2013 [5].

There are a couple of ways to maliciously disable a security manager. **Type confusion** attacks break type safety to craft objects that can perform operations as if they have a different type. Commonly, attackers craft objects that (1) point to the `System` class to directly disable the sandbox or (2) act as if they had the same type as a privileged class loader to elevate a payload class's privileges (see CVE-2012-0507 [18]). In **confused deputy** attacks, exploitative code "convinces" another class to return a reference to a privileged class [19] known to contain a vulnerability that can be attacked to disable the sandbox (see CVE-2012-4681 [20]). The "convincing" is necessary because it is rare that a vulnerable privileged class is directly accessible to all Java applications; doing so violates the *access control* principle that is part of the Java development culture.[4]

Modern exploits that manipulate the security manager simply disable it. This is possible largely because the Java security model grants enormous flexibility to set, weaken, strengthen, or otherwise change a security manager after its creation. Our core thesis is that the overall security of Java applications could be improved by simplifying these security mechanisms, without loss to benign functionality.

3. SECURITY MANAGER STUDY

In this section, we describe the dataset and methodology for our empirical study of the open-source Java application landscape. Our basic research question is: How do benign open-source Java applications interact with the security manager? The answer to this question informs which JVM-level modifications can be used to improve security while maintaining backwards compatibility. There are four possibilities:

1. *Benign applications never disable the security manager.* If true, only exploitative code attempts to disable the security manager, and the ability to do so could be removed from the JVM. This would be easy to implement but would not guard against exploits that weaken the sandbox without disabling it.
2. *Benign applications do not weaken a set security manager.* If true, the JVM could be modified to prevent any weakening or disabling of the sandbox. This is more powerful than simply removing the ability to disable the security manager but is significantly more difficult to implement. For example, if a permission to write to a file is replaced by a permission to write to a different file, is the sandbox weakened, strengthened, or equally secure?
3. *Benign applications never modify the sandbox if a self-protecting security manager has been set.* If true, the JVM could disallow any change to a self-protecting security manager. A runtime monitor in the JVM can determine if a security manager is self-protecting (based on the permission set) when an application attempts to change the sandbox. This is much easier to implement soundly than the previously-described approach, and guards against the same number and types of exploits.
4. *Benign applications do not change a set security manager.* If true, any attempted change to an already established security manager can be considered malicious. This would be the ideal result: restricting this operation is easy to implement in the JVM.

This section describes our study dataset (Section 3.1) and methodology (Section 3.2); we describe results in Section 4.

3.1 Dataset

In the absence of an existing corpus of benign applications that interact with the manager, we combined relevant subjects from the QualitasCorpus (QC) version 20130901 [9], a collection of popular open source Java applications created for empirical studies, and GitHub. While QC contains 112 applications, we found only 24 applications interacted with the security manager. To increase the size and diversity of the dataset (beyond those that meet the QC inclusion requirements), we added 12 applications from Github that also interact with the security manager.[5] Table 2 lists the 36 applications that comprise the dataset. Version numbers and Git commit hashes are available in an online supplement.[6]

We identified relevant applications in the QC set by searching for the keyword `SecurityManager` in the applications' source code. We performed a similar process on the GitHub set, adding the keywords `System.setSecurityManager(` and

[3]Many of the recent vulnerabilities would not have been introduced if the JRE were developed strictly following "The CERT Oracle Secure Coding Standard for Java." [16, 6, 17]

[4]https://blogs.oracle.com/jrose/entry/the_isthmus_in_the_vm

[5]Applets, commonly run in a sandboxed environment, would be natural study subjects. However, we were unable to find any benign applets that interacted with the security manager, likely because of Java's strict restrictions on their behavior.

[6]https://github.com/SecurityManagerCodeBase/ProjectsProfiles/blob/master/projects list.xlsx

Table 2: Security manager interactions dataset.

App name	Description	KLOC	Repo
Apache Ant	Java Project Builder	265	QC
Apache Batik	SVG Image Toolkit	366	QC
Apache Derby	Relational Database	1202	QC
Eclipse	IDE	7460	QC
FreeMind	Mind-Mapping Tool	86	QC
Galleon	Media Server	83	QC
Apache Hadoop	Distrib. Comp. Frwk.	1144	QC
Hibernate	Obj.-Rel. Mapper	376	QC
JBoss	App. Middleware	968	QC
JRuby	Ruby Interpreter	372	QC
Apache Lucene	Search Software	726	QC
Apache MyFaces	Server Software	328	QC
NekoHTML	HTML Parser	13	QC
Netbeans	IDE	8490	QC
OpenJMS	Messaging Service	112	QC
Quartz	Job Scheduler	66	QC
QuickServer	TCP Server Frwk.	64	QC
Spring	Web Dev. Library	828	QC
Apache Struts	Web Dev. Library	277	QC
Apache Tomcat	Web Server	493	QC
Vuze	File Sharing App.	895	QC
Weka	Machine Learning Algs.	531	QC
Apache Xalan	XML Trans. Library	204	QC
Apache Xerces	XML Parsing Library	238	QC
AspectJ	Java Extension	701	GH
driveddoc	Application Connector	7	GH
Gjman	Development Toolkit	<1	GH
IntelliJ IDEA	IDE	4094	GH
oxygen-libcore	Android Dev. Lib.	1134	GH
refact4j	Meta-model Frwk.	21	GH
Security-Manager	Alt. Security Manager	4	GH
Spring-Modules	Spring Extension	212	GH
System Rules	JUnit Extension	2	GH
TimeLag	Sound Application	1	GH
TracEE	JavaEE Support Tool	18	GH
Visor	Closure Library	1	GH

`System.setSecurityManager(null)` to remove false positives and find applications that disable the manager, respectively. We picked the top 6 applications from the results for each keyword, removing manually-identified false positives and duplicates. We studied each GitHub program at its most recent commit and each QC program at its most recent stable release as of June 2014.

3.2 Methodology

We performed a tool-supported manual inspection of the applications in our dataset to group them into qualitative, non-overlapping categories based on their interactions with the security manager. The first category includes applications that can or do *change a set security manager* at run time. If an application did not change the security manager, we looked to see if it set a single security manager during execution. If so, it was categorized as *sets an immutable manager*. Applications that interact with a security manager that they did not set do so to adjust to different security settings. We categorized any such application as *supports being sandboxed*. For the final category, if a program did not contain any interaction with the security manager in the main application, but did interact with it in test cases, we categorized the application as *only interacts in unit tests*; such applications use unit test interactions to test against multiple security settings.

We created static and dynamic analysis tools to assist in a manual inspection of each application's security manager interactions. We created a FindBugs [21] plugin that uses a sound dataflow analysis to determine which manager definitions reach calls to `System.setSecurityManager()`. The dynamic analysis tool uses the Java Virtual Machine Tool Interface (JVMTI) [22] to set a modification watch on the `security` field of Java's `System` class, which stores the security manager object for the application.

We split the dataset between two of the authors, who each analyzed applications using the following steps:

1. Run `grep` on all Java source files in the application to find lines containing the keyword `SecurityManager`. Manually inspect the results in their original source code files to understand how the application interacts with the sandbox.
2. Run the static analysis on retained applications. Manually inspect the returned code, focusing on initialization.
3. Use the dynamic analysis, using parameters informed by the previous steps, to verify conclusions.
4. Summarize operations performed on the security manager, categorize accordingly, and determine if the security manager is self-protecting or defenseless.

We undertook a pilot study where each author independently inspected the same six applications and compared their results. This ensured the two authors understood the analysis steps and produced consistent results.

4. STUDY RESULTS

In this section, we describe the results of our empirical study of open-source Java programs and how they interact with the security manager.

4.1 Summary of benign behaviors

Recall that in Section 3, we refined the high-level research question—how do benign applications interact with the security manager?—into four possibilities, and that the possible mitigations required in each case varied. Revisiting those possibilities with respect to our dataset, we found:

1. Benign applications *do* sometimes disable the security manager. We found that such applications typically use a defenseless sandbox for non-security purposes.
2. Several benign applications *do* provide methods for the user to dynamically change the security policy or the manager in ways that can reduce sandbox security.
3. Benign applications do *not* change the security manager if a self-protecting security manager has been set.
4. Benign applications do sometimes change a set security manager. We observed multiple applications that changed a set security manager.

In terms of the four possible mitigation strategies, only the third—a runtime monitor that blocks modifications to a self-protecting security manager—can improve security without breaking benign behavior. Fortunately, this technique does not require complex, context-sensitive information about whether a change to a policy weakens the sandbox or not.

4.2 Applications by category

Table 3 summarizes our dataset into the categories described in Section 3. The applications in categories 1, 3, and 4 are consistent with any of the potential JVM modifications because they do not interact with the security manager in

Table 3: Classification of application interactions with the security manager.

Type of Interaction	QC	GitHub	Total
1. Sets an immutable manager	6	1	7
2. Changes set manager	5	3	8
3. Supports being sandboxed	10	3	13
4. Interacts only in unit tests	3	5	8

```
691 System.setSecurityManager(new AntSecurityManager(
        originalSM, Thread.currentThread()));
703 // ...execute Ant...
723 finally {
724   // ...
725   if (System.getSecurityManager() instanceof
          AntSecurityManager) {
726       System.setSecurityManager(originalSM);
727   }
```

Figure 2: Snippet of Eclipse code that uses a security manager to prevent Ant from terminating the JVM.

complex ways (i.e., if all applications fell into these categories, the Java security model could be dramatically simplified by eliminating most of its flexibility without breaking existing applications). We will not discuss applications in categories 3 or 4 further, because they did not result in useful insights about common benign behaviors. Most applications in the *Sets an immutable manager* category use the sandbox correctly. We discuss a few particularly interesting examples below. There are eight applications in the *Changes set manager* category, which is the most interesting in terms of possible modifications to the Java security model. They make the most use of Java's flexible security mechanisms. We therefore focus on these applications in our discussion.

We discuss applications that use the sandbox for non-security purposes in Section 4.3 and applications that use the sandbox for its intended security purposes in Section 4.4.

4.3 Non-security uses of the sandbox

Most of the applications that interact with the sandbox in non-security ways did so to enforce architectural constraints when interacting with other applications; the rest forcibly disabled the sandbox to reduce development complexity. This misappropriation of Java's security features increases the difficulty of mitigating attacks against them by increasing the odds of backwards compatibility issues. These applications included applications in both categories 1 and 2, and all require defenseless security managers.

4.3.1 Enforcing architectural constraints

Java applications often call `System.exit()` when an unrecoverable error occurs. When such an application is used as a library, `System.exit()` closes the calling application as well, because both are running in the same JVM. To prevent this without modifying the library application, a calling application needs to enforce the architectural constraint that called library code cannot terminate the JVM.

We found three applications in our dataset that enforce this constraint by setting a security manager that prevents `System.exit()` calls: Eclipse, GJMan, and AspectJ.[7] For example, Eclipse uses Ant as a library, and Ant calls `System.exit()` to terminate a build script in the event of an unrecoverable error. However, when Eclipse uses Ant as a library, it reports an error to the user and continues to execute. Figure 2 shows how Eclipse uses a security manager to enforce this constraint; Eclipse restores the original manager after Ant closes.

This technique does enforce the desired constraint, and appears to be the best solution available in Java at the moment. However, it is problematic for applications using the sandbox for security purposes. The technique requires the application to dynamically change the security manager, which in turn requires a defenseless manager. As a result, the calling applications *themselves* cannot be effectively sandboxed, as might be desirable e.g., when run from Java Web Start. The host machine thus can not be protected from the application itself, or the library code that the application calls.

4.3.2 Web applications outside the sandbox

We found web applications that insist on being run unsandboxed. By default, Java executes such applications inside a self-protecting sandbox with a restrictive policy that excludes operations like accessing local files, retrieving resources from third party servers, or changing the security manager.

Applications in our set that require these permissions opted to run outside of the sandbox. We found two applications that do this: Eclipse and Timelag. Both applications attempted to set the security manager to `null` at the beginning of execution. A restrictive sandbox catches this as a security violation and terminates the application; to run it, the user must ensure that such a sandbox is not set. The rationale for disabling the manager in Eclipse is explained in a code comment that reads, "The launcher to start eclipse using webstart. To use this launcher, the client must accept to give all security permissions." Timelag performs the same operation, but without associated explanatory comments that we could find, thus we can only infer developers motivation.

The developers of Eclipse and Timelag could have either: 1) painstakingly constructed versions of the application that run reasonably using only the permissions available within the sandbox (e.g. by detecting the sandbox and avoiding or disabling privileged operations) or 2) gotten the applications digitally signed by a recognized certificate authority and configured to run with all permissions. These developers likely found these alternatives overly burdensome. The examples from our study suggest that developers are sometimes willing to exchange security guarantees in the interest of avoiding such labor-intensive options.

4.4 Using the security manager for security

Other applications interact with the manager in security-oriented ways. Batik, Eclipse, and Spring-modules allow the user to set and change an existing manager; Ant, Freemind, and Netbeans explicitly set then change the manager.

Batik SVG Toolkit allows users to constrain applications by providing a method to turn the sandbox on or off. This trivially requires a defenseless sandbox. The Batik download page provides several examples of library use, one of which (the "rasterizer" demo) enables and disables the sandbox. However, there seems to be no reason to do so in this case other than to demonstrate the functionality; we were unable

[7] GJMan contains a code comment referencing a blog post that we believe is the origin of this solution. http://www.jroller.com/ethdsy/entry/disabling_system_exit

```
<permissions>
  <grant class="java.security.AllPermission"/>
  <revoke class="java.util.PropertyPermission"
      />
</permissions>
```

Figure 3: An Apache example of an Ant build script element to grant all but one permission. This results in a defenseless security manager; thus revoking one permission does not lead to application security.

to discern the rationale from the examples or documentation.

Ant, Freemind, and Netbeans explicitly set then change the manager, requiring the ability to reconfigure, disable, or weaken the sandbox at runtime to claim backwards compatibility. Ant allows users to create scripts that execute Java classes during a build under a user-specified permissions set. Figure 3 shows an example permission set from the Ant Permissions website.[8] The `grant` element provides the application all permissions, while the `revoke` element restricts the application from using property permissions. Due the use of a defenseless security manager, malicious code can easily disable the sandbox and perform all actions, including those requiring `PropertyPermissions`. Although this policy is only an example, its existence suggests possible confusion on the part of either its author or its consumers about appropriate security policies for untrusted code.

Ant saves the current manager and replaces it with a custom manager before executing constrained external code. The custom manager is not initially defenseless, but contains a private switch to make it so for the purposes of restoring the original manager. Ant therefore catches applications that perform actions restricted by the user while typically protecting sandbox settings. However, it is not clear this implementation is free of vulnerabilities. Netbeans similarly sets a security manager around separate applications.

Both of these cases require a defenseless security manager, otherwise the application would not be able to change the current security manager. Similar to the case in Section 4.3.2, Java provides an "orthodox" mechanism to achieve this goal while aligning with intended sandbox usage: a custom class loader that loads untrusted classes into a constrained protection domain. This approach is more clearly correct and enables a self-protecting sandbox.

An attempt to solve a similar problem in Freemind 0.9.0 illustrates the dangers of a defenseless manager. Freemind is a mind mapping tool that allows users to execute Groovy scripts on such maps (Groovy is a scripting language that is built on top of the JRE). A Java application that executes such a script typically allows it to execute in the same JVM as the application itself. As a result, a specially-crafted mind map can exploit users that run its scripts.

Freemind implements an architecture that is intended to allow the sandbox to enforce a stricter policy on the Groovy scripts than on the rest of Freemind. The design centers around a custom security manager that is set as the system manager in the usual manner. This custom manager contains a field holding a proxy manager for script execution. In this design, all checks to the security manager are ultimately deferred to the proxy manager in this field. When this field is set to `null`, the sandbox is effectively disabled even though

[8]https://ant.apache.org/manual/Types/permissions.html

```
/** By default, everything is allowed. But you
 * can install a different security controller
 * once, until you install it again. Thus, the
 * code executed in between is securely
 * controlled by that different security manager.
 * Moreover, only by double registering the
 * manager is removed. So, no  malicious code
 * can remove the active security manager.
 * @author foltin */
public void setFinalSecurityManager(
     SecurityManager pFinalSM) {
  if(pFinalSM == mFinalSM){
    mFinalSM = null;
    return;
   }
   if(mFinalSM != null) {
     throw new SecurityException("There is a
          SecurityManager installed already.");
   }
   mFinalSM = pFinalSM;
 }
```

Figure 4: Initialization of Freemind's security manager, including a custom proxy. This demonstrates two problems with the sandbox as used by developers: (1) using Java policies as a blacklist is dangerous and (2) modifying the manager at runtime requires a work-around (ineffective or incomplete, in this case) to defend against malicious users.

the system's manager is still set to the custom manager.

Figure 4 shows how Freemind sets the proxy security manager field. Once a manager is set, if `setFinalSecurityManager` is called again with a different security manager, a `SecurityException` is thrown, but calling the method with a reference to the set manager disables the sandbox. The comment implies that this sequence of operations was implemented to prevent malicious applications from changing the settings of the sandbox.

Freemind sets a proxy security manager to stop unsigned scripts from creating network sockets, accessing the filesystem, or executing programs before initiating execution of a Groovy script. The manager grants all other permissions by overriding permission checks with implementations that do nothing, thus any script can turn off the sandbox.

We demonstrated that the custom security manager is easily removed using reflection to show that the problem is more complex than simply fixing permission checks related to setting the security manager. Figure 5 shows a Groovy exploit to turn off the manager. The script gets a reference to the system's manager and its class. The class has the same type as the custom security manager, thus the exploit gets a reference to the proxy manager field. The field is made public to allow the exploit to reflectively `null` it, disabling the sandbox to allow "forbidden" operations. We notified Freemind developers of this vulnerability in August of 2014 and offered our advice in achieving their desired outcome.

All of these applications ran afoul of the Java sandbox's flexibility even though they attempted to use it for its intended purpose. They must all be run with defenseless managers, and those that manipulate the set security policy dynamically do so problematically. While Java does provide the building blocks for constraining a subset of an application with a policy that is stricter than what is imposed on the rest of the application, it is clear that it is too easy to get this wrong: We've seen no case where this goal was achieved

```
def sm = System.getSecurityManager()
def final_sm = sm.getClass().getDeclaredField(
    "mFinalSecurityManager")
final_sm.setAccessible(true)
final_sm.set(sm, null)
new File("hacked.txt").withWriter { out -> out
    .writeLine("HACKED!") }
```

Figure 5: Exploit that breaks out of the scripting sandbox in Freemind to execute arbitrary code.

in a way that is known to be free of vulnerabilities. These case studies support our general claims that the Java security mechanisms are overly complex, and that this complexity contributes to security vulnerabilities in practice.

5. FORTIFYING THE SANDBOX

Based on our study of how open-source Java programs interact with the security manager, we propose two changes to the current Java security model to stop exploits from disabling *self-protecting* managers. These rules can be applied to all Java applications. These rules reduce the flexibility and thus complexity of the Java security model without breaking backwards compatibility in practice:

Privilege escalation rule. If a self-protecting security manager is set for the application, a class may not directly load a more privileged class. This rule is violated when the protection domain of a loaded class implies a permission that is not implied in the protection domain that loaded it.

Security manager rule. The manager cannot be changed if a *self-protecting* security manager has been set by the application. This is violated when code causes a change in the sandbox's configuration, the goal of many exploits.

In this section, we evaluate the protection merits and backwards compatibility of these rules through an implementation of runtime monitors that enforce them. This evaluation was done in collaboration with a large aerospace company. Section 5.1 discusses how we implemented our runtime monitors, Sections 5.2 and 5.3 explain the methodology behind and results of experiments that evaluate the rules' ability to stop exploits while maintaining backwards compatibility.

5.1 Implementation using JVMTI

JVMTI is a native interface that enables the creation of dynamic analysis tools, called agents, such as profilers, debuggers, or thread analyzers. JVMTI agents can intercept and respond to events such as class or thread creation, field access or modification, breakpoints, etc. JVMTI requires turning off the just-in-time compiler (JIT) to enforced the security manager rule, which slows down program execution enough that our monitors are not suitable for adoption with current JVMTI implementations. A more correct and general approach is to embed our rules directly in the JRE. We are currently in communication with the OpenJDK developers on their `security-dev` mailing list about doing so.

5.1.1 Enforcing the privilege escalation rule

To enforce the Privilege Escalation rule, our agent stops a program when a class is loaded to check for privilege escalation. The existence of restricted-access packages complicates this implementation slightly. *Restricted-access packages* are technically public but are intended only for internal JRE use. Benign applications can (and often do) use JDK classes to access these restricted implementations (e.g., much of the functionality in `java.lang.reflect` is backed by the restricted-access `sun` package). We therefore allow the JRE itself to load restricted-access packages at runtime, but prevent such loading from the application classes. Because exploit payloads are not implemented in restricted-access JRE packages, the Privilege Escalation rule can permit this standard behavior while preventing attacks.

Note that there are two ways that application code might directly access such packages in the current security model: (1) exploit a vulnerability in a class that *can* access them or (2) receive permission from the security manager via an `accessClassInPackage("sun")` permission in the policy. The first behavior is undesirable, and is thus rightfully prevented by the enforcement of this rule. The second behavior would require a defenseless manager.

5.1.2 Enforcing the security manager rule

We enforce the Security Manager rule by monitoring every read from and write to the field in the `System` class that stores the security manager (the `security` field). The agent stores a shadow copy of the most recently-set security manager. Whenever the field is written, the agent checks its shadow copy of the manager. If the shadow copy is `null`, the manager is being set for the first time. If the new manager is self-protecting, the agent updates the shadow copy. If not, the agent stops performing stringent checks because the rule does not apply in the presence of a defenseless manager.

The agent checks modifications to the security manager and validates it when it is referenced. The latter is necessary to catch type confusion attacks, which change the manager without triggering a JVMTI modification event. The tool detects unauthorized changes every time the manager is used by comparing the current manager with the shadow copy for changes. Type confusion attacks that masquerade as a privileged class loader will not be detected by our agent, and may still be dangerous when exploited in collaboration with other JRE vulnerabilities.

5.2 Effectiveness at fortifying the sandbox

We evaluated our rules' ability to block sandbox-disabling exploits on ten Java 7 exploits for the browser from Metasploit 4.10.0, an exploit development and delivery framework. We ran the exploits on 64-bit Windows 7 against the initial release of version 7 of the JRE. The ten exploits in question include both type confusion and confused deputy attacks. Metasploit contains many Java exploits outside of the subset we used, but the excluded exploits either only work against long obsolete versions of the JRE or are not well positioned to be used in drive-by downloads. Our results thus show our rules stop the vast majority of current exploits.

We ran the exploits (1) without the agent, (2) with the agent but only enforcing the Privilege Escalation rule, and (3) while enforcing both rules. We tested the Privilege Escalation rule separately because while the Security Manager rule stops all the exploits on its own, the Privilege Escalation rule stops exploits earlier, has significantly less overhead, and can detect attacks that are not explicitly targeting the manager. Table 4 summarizes our results. All ten of the exploits succeeded without the agent. The Privilege Escalation rule stops four of them. All ten were stopped when both rules were enforced.

Together, the rules are capable of stopping current exploit tactics while narrowing available future tactics by blocking

Table 4: Effectiveness test results. The exploits are taken from the subset of Metasploit 4.10.0 that apply in modern environments and follow a drive-by-download paradigm. Taken together, the proposed security model restrictions stop all tested exploits.

	Monitor	
CVE-ID	**Privilege Escalation**	**Both**
2011-3544	Attack Succeeded	Attack Blocked
2012-0507	Attack Blocked	Attack Blocked
2012-4681	Attack Succeeded	Attack Blocked
2012-5076	Attack Succeeded	Attack Blocked
2013-0422	Attack Blocked	Attack Blocked
2013-0431	Attack Blocked	Attack Blocked
2013-1488	Attack Succeeded	Attack Blocked
2013-2423	Attack Succeeded	Attack Blocked
2013-2460	Attack Blocked	Attack Blocked
2013-2465	Attack Succeeded	Attack Blocked

privilege escalation exploit routes.

5.3 Validating Backwards-Compatibility

By construction, the rules do not restrict benign behavior in the applications we studied in Sections 3 and 4. To mitigate the threat of overfitting and increase the generalizability of our results, we also executed the monitors on the applications in Table 5. This set is composed of benign JWS applications that, like applets, are automatically sandboxed. This expands the scope of our results beyond the desktop applications studied above and evaluates our proposed modifications in context (JWS programs are typically run with restrictive security policies). The set also includes closed-source applications, providing evidence that our results generalize beyond open source.

For each program, we confirmed that the agent does not negatively affect benign workloads. ArgoUML, JavaOpenStreetMap, and mucommand contained unit tests that we ran in the presence of our monitors. Costello, and JabRef did not provide tests, but do provide example workloads that we used in their place. CrossFTP contained neither tests nor sample workloads, thus we fuzzed the GUI for 30 minutes using a custom fuzzer and uploaded a file to a remote FTP server as a sample workload.[9]

In each case, we confirmed that the tests, sample workloads, or fuzzed executions worked without a security manager. To sandbox the applications, we developed security policies using a custom security manager that does not throw exceptions and that prints out checked permissions as each program executes. Finally, we ran each case a third time using our policy and the standard Java security manager with our monitors attached and enforcing the rules. The rules did not break any unit tests, sample workloads, or fuzzed executions.

Finally, to validate our rules on representative desktop applications, we confirmed the agent does not break programs in the DaCapo Benchmarks v9.12-bach set [23]. DaCapo systematically exercises each application using a range of inputs to achieve adequate coverage. For all but one case, we set a security manager that granted all permissions and attached our monitors to application execution; we let Batik set its own security manager because it exits if it cannot do so. Our rules did not break any DaCapo applications.

[9]GUI fuzzing source code can be found at https://goo.gl/ccTLVR.

6. LIMITATIONS AND VALIDITY

Limitations. Neither of the rules we propose in Section 5 will stop all Java exploits. While the rules catch all of the exploits in our set, some Java vulnerabilities can be exploited to cause significant damage without disabling the security manager. For example, our rules will not detect type confusion exploits that mimic privileged classes to perform their operations directly. However, our rules substantially improve Java sandbox security, and future work will be able to build upon these results to create mitigation techniques for additional types of exploits.

Internal validity. Our study results are dependent on accurately studying the source code of applications and their comments. In most cases, security manager interactions are easily understood, but there are a few particularly complex interactions that may be misdiagnosed. Furthermore, we did not review all application code, thus we may have taken a comment or some source code out of context in larger applications. Finally, using two different reviewers may lead to variations in the interpretations of some of the data.

We mitigated these threats by using a checklist, FindBugs plugin, and JVMTI agent to provide reviewers consistent processes for reviewing code and validating their results. Furthermore, we inspected entire source files that contained security manager operations. We tested our tools and processes in a pilot study to find and mitigate sources of inconsistencies.

External validity. The study only includes open-source applications. It is possible that closed-source applications interact with the security manager in ways that we did not see in the open-source community. However, we inspected a few small closed-source applications with our aerospace collaborators. We did not find any code that suggested this is the case. This result is further supported by the closed-source programs included in the dataset in Section 5.3.

Reliability. While the majority of the study is easily replicable, GitHub search results are constantly changing. Using GitHub to generate a new dataset would likely result in a different dataset. Furthermore, over the course of the study, one application either became a private repository or was removed from GitHub (Visor).

7. RELATED WORK

As far as we are aware, no study has examined Java applications' use of the sandbox. However, several recent studies have examined the use of security libraries that can be overly complex or misused, discovering rampant misuse and serious vulnerabilities. Georgiev *et al.* uncovered vulnerabilities in dozens of security critical applications caused by SSL library protocol violations [24]. These applications misconfigured high-level libraries such that the high-level libraries misused low-level SSL libraries, which in turn failed silently. Somorovsky *et al.* demonstrate vulnerabilities in 11 security frameworks such that Security Assertion Markup Language (SAML) assertions are not checked properly in the face of certain API mis-orderings [25]. Li *et al.* examined browser-based password managers and found that many of their features relied on an incorrect version of the same-origin policy, which could allow attackers to steal user credentials [26].

Our rules increase the security of the sandbox by effectively removing unnecessary features. Prior work has taken a different approach, proposing to re-implement the Java sandbox or add to it to increase security. Cappos *et al.* cre-

Table 5: Backwards compatibility dataset.

Name	Description	KLOC	Workload	Latest Commit
ArgoUML	UML Tool	389	1244 test cases	1/11/15
Costello	GUI Testing Frontend	closed source	9 provided examples	5/09/12
CrossFTP	FTP Client	closed source	GUI fuzzing, sample workload	1/18/15
JavaOpenStreetMap	Map Editor	343	406 test cases	1/18/15
JabRef	Reference Manager	148	3 provided examples	3/11/14
mucommander	File Manager	106	27 test cases	1/23/14

ated a new sandbox structure involving a security-isolated kernel separating sandboxed applications from the main system [27]. They validated this structure by translating past Java CVEs into exploits for the new kernel. Provos *et al.* describe a method of separating privileges to reduce privilege escalation [28]. Their approach is partially implemented in the Java security model. Li and Srisa-an extended the Java sandbox by providing extra protection for JNI calls [29]. Their implementation, Quarantine, separates JNI accessible objects to a heap which contains extra protection mechanisms. The performance of their mechanism is also measured using DaCapo. Siefers *et al.* created a tool, Robusta, which separates JNI code into another sandbox [30]. Sun and Tan extend the Robusta technique to be JVM independent [31].

Java applets are the most common ways to transmit Java exploits. Detectors have been created to identify drive-by downloads in JavaScript [32], and in Adobe Flash [33]. Helmer *et al.* used machine learning to identify malicious applets [34]. Their approach monitored system call traces to identify malicious behavior after execution. However, this approach is entirely reactive. Our approach terminates exploits when they attempt to break out of the sandbox, before they perform their payloads. Schlumberger *et al.* used machine learning and static analysis to identify common exploit features in malicious applets [35]. Blasing *et al.* detect malicious Android applications using static and dynamic analyses of sandboxed executions [36]. Unlike these automated approaches, our rules show a better understanding of benign sandbox interactions can inform unique mitigation strategies.

8. CONCLUSION

Li Gong, the primary designer of the Java security architecture, admitted in a ten year retrospective on Java Security that he did not know how or how extensively the "fine grained access control mechanism" (i.e. the Java sandbox) is used [37]. Our study fills in that gap.

Our empirical study of open-source applications supports the hypothesis that the Java security model provides more flexibility than developers use in practice. The study also strongly suggests that the model's complexity leads to unnecessary vulnerabilities and bad security practices. We further validated the findings of our study by defining two rules, which together successfully defeated Metasploit's applet exploits without breaking backwards compatibility.

We take several general lessons from these findings. First, Java should provide simpler alternative mechanisms for various common goals, such as constraining access to global resources or adapting to multiple security contexts. We found that developers sometimes use the sandbox to prevent third party components from calling `System.exit()`, a specific instance of a more general development problem: frameworks often need to enforce constraints on plugins (e.g., to ensure non-interference). We also observed that developers who attempted to make non-trivial use of the sandbox often do so incorrectly, even though the functionality in question could theoretically be implemented correctly within the current model (albeit with increased complexity). One promising approach is to allow programmers to temporarily strengthen security policies (e.g. by adding a permission).

We observed evidence that many developers struggle to understand and use the security manager for any purpose. This is perhaps why there were only 36 applications in our sample. Some developers seemed to misunderstand the interaction between policy files and the security manager that enforces them. Others appear confused about how permissions work, not realizing that restricting just one permission but allowing all others results in a *defenseless* sandbox. Our concerns here are shared by the IntelliJ developers, who include static analysis checks to warn developers that a security expert should check all security manager interactions [38]. In general, sandbox-defeating permissions should be packaged and segregated to prevent accidentally defenseless sandboxes. Finally, some developers appear to believe the sandbox functions as a blacklist when, in reality, it is a whitelist. In total, our results and observations suggest that the model itself should be simplified, and that more resources—tool support, improved documentation, or better error messages—should be dedicated to helping developers correctly use the sandbox.

9. ACKNOWLEDGEMENTS

This material is based upon work supported by the U.S. Department of Defense through the Office of the Assistant Secretary of Defense for Research and Engineering (ASD(R&E)) under Contract HQ0034-13-D-0004, the National Security Agency under Lablet Contract H98230-14-C-0140, and the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1252522. 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 ASD(R&E), NSA, or NSF.

10. REFERENCES

[1] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2.," in *USENIX Symposium on Internet Technologies and Systems*, pp. 103–112, 1997.

[2] L. Gong and G. Ellison, *Inside Java (TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.

[3] IBM Security Systems, "IBM X-Force threat intelligence report." http://www.ibm.com/security/xforce/, February 2014.

[4] L. Garber, "Have Java's Security Issues Gotten out of Hand?," in *2012 IEEE Technology News*, pp. 18–21,

2012.

[5] A. Singh and S. Kapoor, "Get Set Null Java Security." http://www.fireeye.com/blog/technical/2013/06/get-set-null-java-security.html, June 2013.

[6] D. Svoboda, "Anatomy of Java Exploits." http://www.cert.org/blogs/certcc/post.cfm?EntryID=136.

[7] A. Gowdiak, "Security Vulnerabilities in Java SE," Tech. Rep. SE-2012-01 Project, Security Explorations, 2012.

[8] J. W. Oh, "Recent Java exploitation trends and malware," Tech. Rep. BH-US-12, Black Hat, 2012.

[9] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *Asia Pacific Software Engineering Conference (APSEC)*, pp. 336–345, Dec. 2010.

[10] "Permissions in the JDK." http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html, 2014.

[11] "Default Policy Implementation and Policy File Syntax." http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html.

[12] A. Banerjee and D. A. Naumann, "Stack-based access control and secure information flow," *Journal of Functional Programming*, vol. 15, pp. 131–177, Mar. 2005.

[13] F. Besson, T. Blanc, C. Fournet, and A. Gordon, "From stack inspection to access control: A security analysis for libraries," in *Computer Security Foundations Workshop*, pp. 61–75, June 2004.

[14] D. S. Wallach and E. W. Felten, "Understanding Java Stack Inspection," in *IEEE Symposium on Security and Privacy*, pp. 52–63, 1998.

[15] C. Fournet and A. D. Gordon, "Stack Inspection: Theory and Variants," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 307–318, 2002.

[16] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, and D. Svoboda, *The CERT Oracle Secure Coding Standard for Java*. SEI Series in Software Engineering, Addison-Wesley Professional, 1st ed., Sept. 2011.

[17] D. Svoboda and Y. Toda, "Anatomy of Another Java Zero-Day Exploit." https://oracleus.activeevents.com/2014/connect/sessionDetail.ww?SESSION_ID=2120, Sept. 2014.

[18] "Vulnerability Summary for CVE-2012-0507." https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-0507, June 2012.

[19] N. Hardy, "The Confused Deputy: (or Why Capabilities Might Have Been Invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, pp. 36–38, Oct. 1988.

[20] "Vulnerability Summary for CVE-2012-4681." http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-4681, Oct. 2013.

[21] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, pp. 92–106, Dec. 2004.

[22] "Java Virtual Machine Tool Interface." https://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/.

[23] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pp. 169–190, Oct. 2006.

[24] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: Validating SSL certificates in non-browser software," in *ACM Conference on Computer and Communications Security (CCS)*, pp. 38–49, ACM, 2012.

[25] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen, "On breaking SAML: Be whoever you want to be," in *USENIX Security*, pp. 21–21, 2012.

[26] Z. Li, W. He, D. Akhawe, and D. Song, "The emperor's new password manager: Security analysis of web-based password managers," in *USENIX Security*, 2014.

[27] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson, "Retaining sandbox containment despite bugs in privileged memory-safe code," in *ACM Conference on Computer and Communications Security (CCS)*, pp. 212–223, ACM, 2010.

[28] N. Provos, M. Friedl, and P. Honeyman, "Preventing Privilege Escalation," in *USENIX Security*, 2003.

[29] D. Li and W. Srisa-an, "Quarantine: A Framework to Mitigate Memory Errors in JNI Applications," in *Conference on Principles and Practice of Programming in Java (PPPJ)*, pp. 1–10, 2011.

[30] J. Siefers, G. Tan, and G. Morrisett, "Robusta: Taming the Native Beast of the JVM," in *ACM Conference on Computer and Communications Security (CCS)*, pp. 201–211, 2010.

[31] M. Sun and G. Tan, "JVM-Portable Sandboxing of Java's Native Libraries," in *European Symposium on Research in Computer Security (ESORICS)*, pp. 842–858, 2012.

[32] M. Cova, C. Kruegel, and G. Vigna, "Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code," in *International World Wide Web Conference (WWW)*, pp. 281–290, 2010.

[33] S. Ford, M. Cova, C. Kruegel, and G. Vigna, "Analyzing and Detecting Malicious Flash Advertisements," in *Annual Computer Security Applications Conference (ACSAC)*, pp. 363–372, 2009.

[34] G. Helmer, J. Wong, and S. Madaka, "Anomalous Intrusion Detection System for Hostile Java Applets," *J. Syst. Softw.*, vol. 55, pp. 273–286, Jan. 2001.

[35] J. Schlumberger, C. Kruegel, and G. Vigna, "Jarhead Analysis and Detection of Malicious Java Applets," in *Annual Computer Security Applications Conference (ACSAC)*, pp. 249–257, 2012.

[36] T. Blasing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Conference on Malicious and Unwanted Software (MALWARE)*, pp. 55–62, 2010.

[37] L. Gong, "Java security: a ten year retrospective," in *Annual Computer Security Applications Conference (ACSAC)*, pp. 395–405, 2009.

[38] "IntelliJ IDEA inspections list (632)." http://www.jetbrains.com/idea/documentation/inspections.jsp.