Hardening Java's Access Control by Abolishing Implicit Privilege Elevation

Philipp Holzinger*, Ben Hermann†, Johannes Lerch†, Eric Bodden‡ and Mira Mezini†
*Fraunhofer SIT, Email: philipp.holzinger@sit.fraunhofer.de
†Technische Universität Darmstadt, Email: {lastname}@cs.tu-darmstadt.de
‡Fraunhofer IEM & Paderborn University, Email: eric.bodden@uni-paderborn.de

***Abstract*—While the Java runtime is installed on billions of devices and servers worldwide, it remains a primary attack vector for online criminals. As recent studies show, the majority of all exploited Java vulnerabilities comprise incorrect or insufficient implementations of access-control checks. This paper for the first time studies the problem in depth. As we find, attacks are enabled by shortcuts that short-circuit Java's general principle of stack-based access control. These shortcuts, originally introduced for ease of use and to improve performance, cause Java to elevate the privileges of code implicitly. As we show, this creates many pitfalls for software maintenance, making it all too easy for maintainers of the runtime to introduce blatant confused-deputy vulnerabilities even by just applying normally semantics-preserving refactorings.**

How can this problem be solved? Can one implement Java's access control without shortcuts, and if so, does this implementation remain usable and efficient? To answer those questions, we conducted a tool-assisted adaptation of the Java Class Library (JCL), avoiding (most) shortcuts and therefore moving to a fully explicit model of privilege elevation. As we show, the proposed changes significantly harden the JCL against attacks: they effectively hinder the introduction of new confused-deputy vulnerabilities in future library versions, and successfully restrict the capabilities of attackers when exploiting certain existing vulnerabilities. We discuss usability considerations, and through a set of large-scale experiments show that with current JVM technology such a faithful implementation of stack-based access control induces no observable performance loss.

I. Introduction

The Java platform is installed and running on literally billions of devices and servers worldwide [1]. It is also one of the first execution environments to feature an elaborate security model [2]. The platform was designed with the explicit requirement for the secure execution of code retrieved from untrusted locations such as applets on a website that will run in the client's browser. Yet, according to Cisco's Annual Security Reports Java was the number one attack vector for web exploits in 2013 with a share of 87% [3], and even 91% in 2014, thus clearly outranking Flash and Adobe PDF [4].

A large variety of attacks was enabled due to incorrect or insufficient implementations of access control checks. In particular, Holzinger et al. recently showed in a large-scale study on more than ten years of Java exploitation [5] that the by far most prominent attack vectors exploit vulnerabilities caused by an implicit assignment and elevation of privileges within the Java Class Library (JCL). In this work, we investigate this prevalent problem in full depth and suggest and evaluate a concrete mitigation strategy. The goal is not just to significantly harden the Java platform but to also draw important conclusions for the secure design of future runtimes.

At a first glance, the implicit assignment of privilige seems to ease the life of JCL developers, as it allows them to access security sensitive low-level operations without explicit access-control checks. As our research shows, though, this advantage is greatly outweighed by a severe drawback of such an implicit privilege elevation: if developers do not—at all times—properly protect the privileges they are assigned, they might accidentally leak them to attackers, opening up the runtime to so-called confused-deputy attacks [6]. But due to the implicitness of the privilege elevation developers are most often unaware of having obtained privileges in the first place, and hence also unaware of their obligation to protect them.

On a lower level, the Java Security Model features isolated zones where code can run with limited privileges such as a restricted access to the file system. For any given Java Virtual Machine (JVM), administrators can configure this JVM's security setting through a specialized policy language. A set of standard policies, shipped with the Java runtime, provides default protection domains, for instance for applets and applications using Java Web Start. During runtime, the JVM uses stack-based access control [2] to check if a caller has the permission to access any given security-sensitive functionality. In theory, the JVM performs a stack walk, checking that each and every frame on the current call stack is associated with sufficient access permissions. In cases where one of those frames belongs to an untrusted applet, for instance, this check will fail, resulting in a `SecurityException` being thrown.

But as we find, this is only theory. In practice, it shows that many security-sensitive methods in the Java Class Library (JCL) implement what we call *shortcuts*: They execute stack walks only under certain circumstances and use heuristics (such as checking the immediate caller's classloader) to validate the secure execution in other cases. Methods with shortcuts are generally caller-sensitive: Depending on the nature of the shortcut, they grant privileges implicitly to certain groups of callers, in many cases to all callers within the JCL.

As we find, shortcuts are highly problematic for two reasons. First, they pose a significant risk to the security of the overall Java Platform, due to the fragile nature of caller-sensitive behavior. As demonstrated by previous exploits, attackers can abuse insecure use of reflection to invoke shortcut-

DOI 10.1109/SP.2017.16

containing methods, which help them break out of Java's sandbox. As Holzinger et al. showed, caller sensitivity in combination with confused deputies alone is abused by 36% of all exploits they found in the wild [5].

Second, shortcuts severely impede the maintainability of the Java runtime's implementation. During our investigation we found several places inside the runtime library, at which developers could inadvertently break the entire platform's security through simple code transformations that would otherwise be considered semantics-preserving refactorings. In particular this is true for the introduction of wrapper methods. Wrappers modify the call stack, which can inadvertently cause the shortcuts to check properties of the wrong stack frames. Incidentally, some of those places were even commented with warnings to "NOT REFACTOR THIS CODE". In addition to cases, where a shortcut poses an immediate threat, there are often cases in which it can lead to an exploitable vulnerability later, due to simple code maintenance/evolution.

We conducted an experiment, transforming a JCL release such that explicit `doPrivileged`-calls become the *only* way in which the JCL elevates privileges. This has several advantages. First, as we elaborate later, it eliminates certain attack vectors that abuse insecure use of reflection to profit from shortcuts. Second, it makes privilege elevation *explicit*, which eliminates the potential to elevate privileges accidentally through code restructuring/evolution. Third, explicit privilege elevation allows both security experts and code analysis tools [7], [8] to focus on `doPrivileged`-calls to ensure the security of the access-control implementation.

One prevalent reason for introducing shortcuts in the first place is that stack-based access control is expensive (after all, the JVM needs to reify the call stack); shortcuts lead to a faster implementation of access control [9]. In this work we show through a set of large-scale experiments that no such penalty is measurable on the DaCapo benchmark suite [10], despite the fact that it makes heavy use of security-sensitive APIs, and also state reasons for why this is the case.

A second reason for the presence of shortcuts is that the implicit assignment of privilege is convenient, as it reduces the need to elevate priviliges explicitly, e.g. through an appropriate access-control policy. Another contribution of this paper is thus a detailed assessment on the usability implications that a move from implicit to purely explicit privilege elevation entails. This assessment allows us to provide specific guidance for an actual implementation of our hardening in Java's codebase. Last but not least we discuss lessons learned that ought to guide design decisions in the security architecture of future language runtimes.

To summarize, this work makes the following contributions:

- the first detailed analysis of the effects of implicit privilege elevation and shortcuts for access-control checks in Java, along with the security and maintainability problems they induce (Section III),
- a tool-assisted analysis and adaptation technique to avoid the risk of (introducing) confused deputies in the JCL due to shortcuts (Section IV),
- an adapted version of the JCL that implements access control without shortcuts, a detailed explanation of why this adapted version enhances security and maintainability (Section IV-D),
- a set of large-scale experiments showing that this added security and maintainability comes at a negligible runtime cost (Section V), and
- guidance on the productive use of our proposed solution, and an outline of open research questions (Section VI), as well as general lessons learned from our in-depth analysis (Section VII).

All artifacts needed to reproduce our results are publicly available.[1]

II. Background

The JCL restricts access to security-sensitive resources by means of security-policy enforcement. Only code with appropriate permissions may use, e.g., filesystem or network functionality. To this end, every security-sensitive operation is guarded by a call to the security manager. The security manager applies a stack-based access-control algorithm to decide whether attempted access shall be granted or denied. Permission checks are performed by inspecting the current call stack and computing the intersection of the permissions that the declaring class of each method on the stack has been assigned by the running virtual machine's security policy. If the required permission is contained in the intersection, access is granted by returning from the check method, otherwise an exception is thrown that prevents the attempted action.

There are two deviations from this basic model: (a) privileged actions [2] and (b) what we call shortcuts.

Privileged actions

Code with appropriate permissions can explicitly elevate privileges for specific operations by a call to `doPrivileged`. This ensures that subsequent access-control decisions ignore all callers on the call stack before the `doPrivileged`-call. This concept enables trusted code to act as a guarantor on whose behalf untrusted code may perform a certain action. Trusted code on whose behalf the action is performed has to ensure that all security-sensitive actions performed in this context cannot cause harm even if triggered by malicious code. Consider for illustration Listing 2, where `readProp` uses `doPrivileged` to temporarily elevate the privileges of the executing thread such that the call to `checkPermission` in `openFile` can succeed. In the example, the "privileged" call to `openFile` is explicitly entrusted not to misuse its privileges, in this case rightfully so, as `readProp` uses the privilege carefully, reading only the system-properties file it needs, exposing no file handle to a potential attacker.

[1] https://github.com/stg-tud/jdeopt

```
1  class FileAccess {
2    File openFile(String path) {
3      //if no trusted library class
4      Class c = Reflection.getCallerClass();
5      if(c!=null && ClassLoader.getClassLoader(c)!=null) {
6        SecurityManager s = System.getSecurityManager();
7        s.checkPermission(new FilePermission(path));
8      }
9      return newFileHandle(path);
10   }
11 }
12 class SystemProperties {
13   public String readProp(String name) {
14     File f = FileAccess.
15       openFile(JDK_PATH+"/system.properties");
16     ... //read property
17   }
18 }
19
20 // code below this point is added in a later release
21 class Util {
22   public File openFileFromRoot(String name) {
23     return FileAccess.openFile("/"+name);
24   }
25 }
```

Listing 1. Example shortcut for permission check

Shortcut checks

We say that a method contains a *shortcut* if it contains a permission check, i.e., a call to a method of the form `SecurityManager.check*`, that is carried out only if certain constraints on the current call stack are satisfied. These constraints are expressed through conditionals and typically take the immediate caller and/or its classloader into account. `Class.getDeclaredMethods` is an example for such a method. It skips a permission check in case the immediate caller was defined by the same classloader as the class whose members shall be accessed by the call. The Secure Coding Guidelines for Java [11] (JSCG) list a number of such "caller-sensitive" methods [9] in sections 9.8 through 9.11. They should be used with special care to avoid the introduction of vulnerabilities. Only a subset of those methods use their knowledge about the call stack to implement shortcuts.

For illustration, consider the simplified example in Listing 1. Assume that classes `FileAccess` and `SystemProperties` exist in some release of the JCL and class `Util` has been introduced in a later release to the (trusted) library, as a convenience. Method `openFile` opens arbitrary files on the caller's behalf. Since this is a security-sensitive operation, the method checks for the appropriate `FilePermission`. However, in doing so, it takes a shortcut: It performs the permission check actually only for such callers that are not associated with a `null` classloader (see line 5). All classes in the JCL, including `FileAccess` and `SystemProperties` here, are associated with the classloader `null`, i.e., by taking the shortcut, the method `openFile` *implicitly trusts* all calls from the JCL.

This is no problem with callers that actually deserve this trust. For instance, the method `readProp` uses the privilege carefully, reading only the system-properties file it needs and otherwise exposing no file handle. However, it is fairly easy to accidentally expose the elevated privilege to untrusted users. For instance, to the developer of the new class `Util`, it is not at all obvious that the introduction of such a simple wrapper could have severe security implications. In the example, the new method `openFileFromRoot` is an example of a confused deputy: it exposes the complete behavior of `openFile` to its callers, without any filtering, checking, or sanitization of the passed arguments. In this way, clients outside the trusted base can misuse `Util` to bypass all permission checks within `openFile`, as `Util` is trusted, i.e., associated with a `null` classloader. In the past, accidentally introduced confused deputies like `Util` have actually led to severe vulnerabilities in the JCL that allowed attackers to completely break out of the JVM's sandbox (e.g., CVE-2012-5088).

In this work we propose a systematic tool-assisted hardening of the JCL that virtually avoids this class of security-breaking programming mistakes. On a high level of abstraction, the hardening causes the JCL to make privilege elevation *explicit* in (almost) all cases. Our approach works in two steps. An initial, very lightweight static analysis step assists in locating shortcuts in permission checks like the one illustrated in Listing 1, line 5. A subsequent adaptation step then transforms the JCL such as to avoid the possibility to accidentally introduce confused deputies like `Util`, while retaining backward compatibility. The adaptation eliminates the shortcuts by introducing proper permission checks in every instance (with two interesting exceptions described later) via Java's `doPrivileged`-wrappers. By calling a `doPrivileged`-wrapper, a piece of code can elevate a caller's privileges *temporarily and explicitly*, vouching for the correctness and security of the actions performed on the caller's behalf. Our adapted JCL uses `doPrivileged`-wrappers to elevate privileges *explicitly* where they were previously elevated implicitly by shortcuts. This retains backward compatibility, meaning that all applications that were designed and compiled for the original JCL also run on the modified JCL.

For illustration Listing 2 shows the result of applying our adaptation to the code in Listing 1. The adapted version of `FileAccess.openFile` does not take a shortcut anymore, causing a stack walk in every instance initiated by the call to `checkPermission`. An unprivileged attacker calling `Util.openFileFromRoot` will cause the permission check to fail, because the triggered stack walk recognizes the unprivileged attacker and throws a `SecurityException`, thus eliminating the previous vulnerability. To make adaptation backward compatible, the adapted `readProp` explicitly elevates its privileges through a `doPrivileged` call (lines 11–13).

After applying the proposed technique to a JCL release, `doPrivileged`-wrappers become the *only* way in which the JCL elevates privileges. As we show later, this greatly reduces the runtime's attack surface.

III. Problem Statement

This section elaborates on the problems with shortcuts in Java's permission checks. Shortcuts implicitly elevate privi-

```
1  class FileAccess {
2    File openFile(String path) {
3      SecurityManager s = System.getSecurityManager();
4      s.checkPermission(new FilePermission(path));
5      return newFileHandle(path);
6    }
7  }
8  class SystemProperties {
9    public String readProp(String name) {
10     // Java 8's lambda syntax ...
11     File f = AccessController.doPrivileged(
12       (PrivilegedAction<File>) () -> FileAccess.
13         openFile(JDK_PATH+"/system.properties"));
14     ... //read property
15   }
16 }
17
18 // code below this point is added in a later release
19 class Util {
20   public File openFileFromRoot(String name) {
21     //will throw SecurityException
22     return FileAccess.openFile("/"+name);
23   }
24 }
```

Listing 2. Example of adapted code

leges to a certain subset of callers, with two severe effects.

On the one hand, shortcuts increase the number of potential attack vectors. Attackers can abuse reflection to call shortcut-containing methods on behalf of a trusted class. Many of these methods will skip a permission check, because the immediate caller is trusted, and thus provide functionality that was intended to be restricted. Two examples of such methods that are known to be of great value to attackers are `Class.getDeclaredFields` and `Class.getDeclaredMethods`, which skip permission checks, if the immediate caller is defined by the same classloader as the class whose members shall be accessed by the call. They can be used to access private members of a class that were intended to be inaccessible by untrusted code. To find examples of such kinds of attacks, we manually reviewed a sample set of exploits that was provided by Security Explorations [12] consisting of 48 original exploits. We found that at least four of those exploits depend on shortcuts.[2] As a recent study by Holzinger et al. shows, however, the problem is even more prevalent: that caller sensitivity in combination with confused deputies alone is abused by 36% of all exploits they found in the wild [5].

On the other hand, the potential is high that developers, who are either not aware of or unable to properly reason about the implicitly elevated privileges, introduce security flaws when extending the library by implementing new callers of methods with shortcuts or evolve existing code. In the following, we elaborate on the reasons.

First, information about shortcuts is rarely part of the method's documentation. Hence, developers of any caller methods will not be aware that calling certain methods imposes requirements on their implementation to not expose critical functionality to untrusted code. Consider again `Class.getDeclaredMethods` and the scenario, where a maintainer of the JCL introduces a wrapper, `MethodFilters` whose `privateMethodsOf(Class)` calls `Class.getDeclaredMethods` and filters out the non-private methods from the set returned by it. This seemingly harmless new functionality allows attackers to access *all private methods of all classes within the JCL*. The shortcut within `Class.getDeclaredMethods` only considers `MethodFilters`'s classloader, which does coincide with that of `Class`, but let go unchecked the potentially attacker-controlled caller of `MethodFilters`.

In the best case, the developer of a caller method knows about the shortcut in the callee, e.g., through the Java Secure Coding Guidelines (JSCG), which provides explanations and a list of methods that implement shortcuts. He may consciously make the decision to take the risk and the responsibility to prevent harm. When he does so, this decision is not documented in the code. In future code revisions, maintainers unaware of the special requirements imposed by the shortcut may inadvertently invalidate the security precautions taken by the original author.

Second, hardcoded shortcuts are hard to analyze. There is no dedicated Java language construct or API support to express and document assumptions about the call stack. As a result, the effect and the scope of the implicit privilege elevation can only be reasoned about by careful examination of the shortcut's implementation and in addition requires deep knowledge of JCL classes and their properties. This reasoning is a very tedious and error-prone task. Thus, even when the developers know the list of methods that implement shortcuts by heart, using them implies constant awareness and a lot of effort by developers to prevent the introduction of new confused-deputy vulnerabilities.

Third, it is hard to maintain the security of shortcut-containing code in the face of code evolution. Security-sensitive methods that implement shortcuts often assume a specific order of callers on the call stack. Changes to the code that affect the order of callers may cause the sensitive method to misbehave, if assumptions are not properly adjusted. It is hard to judge whether a local change in the code base violates the assumptions of some hardcoded shortcut. Thus, every change has to be properly analyzed to rule out potentially negative side effects on policy enforcement. Since, as already mentioned, such an analysis is manual and very involved, the risk is high that code evolution will introduce vulnerabilities.

Fourth, hardcoded shortcuts are inflexible. Changes in the deployment environment for Java applications may affect risk considerations and security requirements. Adjusting policies accordingly is a matter of configuration, whereas changing hardcoded shortcuts is impractical.

Shortcuts violate several well-accepted secure design principles. Yee proposed [13] a set of ten fundamental principles that should be followed when designing a secure system. While those principles were originally developed to reason about the usability of entire software systems from an end-user's perspective (e.g., the user interface of a password prompt), Türpe showed in [14] that the same principles can

[2] Issues 32 (using CVE-2012-5088), 35, 36, 37

```
void checkMemberAccess(Class clazz, int w) {
  if (w != Member.PUBLIC) {
    Class stack[] = getClassContext();
    /* stack depth of 4 should be the caller
     * of one of the methods in java.lang.Class
     * that invoke checkMember access.
     * The stack should look like:
     * someCaller                        [3]
     * java.lang.Class.someReflectionAPI [2]
     * java.lang.Class.checkMemberAccess [1]
     * SecurityManager.checkMemberAccess [0] */
    if ((stack.length<4) ||
      (stack[3].getClassLoader() !=
        clazz.getClassLoader())) {
      checkPermission(CHECK_MEMBER_ACCESS);
    }
  }
}
```

Listing 3. Shortcut permission check with inline comments documenting assumptions about callers in `java.lang.SecurityManager` (Java 1.7.0u25)

also be applied for purposes of API usability evaluation. The deficiencies discussed above violate five of the ten principles: (a) "Path of Least Resistance", because developers need extra effort to prevent the introduction of confused-deputies; (b) "Explicit Authority", due to the implicit nature of shortcuts; (c) "Visibility", since shortcut-containing methods appear as "regular" methods; (d) "Revocability", because developers cannot refrain from privilege elevation through shortcuts; and finally (e) "Clarity", because the effects on policy enforcement are unclear when using a method that contains a shortcut.

To illustrate an extraordinary case of shortcuts, Listing 3 shows actual code that was released as part of Java 1.7.0 update 25. The method in this example was used, for instance, by `java.lang.Class`, to restrict reflective access from one class to members of another class. A shortcut will bypass a call to `checkPermission`, thus preventing stack inspection and granting the privilege implicitly if certain constraints on the call stack are unsatisfied. This is an interesting case because `checkMemberAccess` makes extensive assumptions about the call stack, involving the size of the stack and the order of callers. It may easily happen that code will be introduced that violates these assumptions, which is also why one finds the following warning in `java.lang.Class`: "Don't refactor; otherwise break the stack depth [...] as specified.". Already in 2009, Li Gong underpinned that counting stack frames is highly fragile and highlighted stack inspection as a key feature of Java 1.2 that would finally allow for more reliable access-control checks [15].

To recap the discussion, we conclude that shortcuts significantly complicate the task of writing secure code in the first place and even more so the task of maintaining security in the face of evolution. This claim is supported by various confused-deputy vulnerabilities in past versions of Java, which demonstrated how attackers can profit from inadvertently exposed functionality. The most prominent of these vulnerabilities are listed in the NVD [16] under CVE-2012-4681, CVE-2012-5088, CVE-2013-0422, and CVE-2013-2460.

Privileged actions versus shortcuts

Since our approach to address the discussed problems with shortcuts is to replace them by privileged actions, we conclude this section by briefly considering privileged actions and shortcuts side-by-side.

Privileged actions are in many ways similar to shortcuts. They terminate stack walks early, thus potentially allowing untrusted code to perform security-sensitive actions on behalf of trusted code. In this sense, all callers of methods that implement shortcuts are in the same intermediary role as code executing within `doPrivileged`. In both situations, developers have to ensure that security-sensitive functionality is not exposed in a way that is profitable to attackers. However, besides the above similarities, there are significant differences between privileged actions and shortcuts. Using a privileged action involves a developer who actively declares to make the conscious decision to take and control a risk, and who can therefore be assumed to know that security precautions are required. Calling `doPrivileged` makes this decision explicit. The fact that privileged actions are explicitly marked as such and restricted by a lexical scope makes them easy to reason about. Unlike implicit shortcuts, the use of privileged actions is supported by a dedicated, well-specified API, and well-defined algorithms, e.g., the access-control algorithm as documented in [2]. Automatic program analysis, as well as manual reviewers benefit from this dedicated support.

IV. Proposed Solution

Our proposed solution comprises three steps. First, one has to locate all shortcuts. Note that there is no complete documentation on shortcuts available. The JSCG is helpful because it provides a list of officially supported caller-sensitive methods. However, it does not state which of these methods implement an access-control shortcut, and the list is not guaranteed to be complete today nor in the future. The second step is to remove the shortcuts found. Finally, for backward compatibility, one has to wrap the *calls* in the JCL to those methods that formerly implemented shortcuts into privileged actions.

We implemented our proposed solution on the basis of OpenJDK 8 b132-03_mar_2014, such that we can evaluate its feasibility and performance impact. We applied a semi-automated approach to locate and remove shortcuts in the JCL. The following three sections provide details on each step of the transformation process. We expose all artifacts required to reproduce our results with this paper.[3]

A. Locating shortcuts

The identification of JCL methods that contain shortcuts is complicated by four related factors. First, there is no dedicated language support to express constraints on the call stack, which is why they cannot be trivially recognized. Second, security-sensitive methods do not necessarily implement shortcuts and calls to the security manager by themselves, but may use

[3]https://github.com/stg-tud/jdeopt

helper methods instead. Third, security-sensitive methods are scattered all over the code base, so the identification process has to take into account all parts of the JCL. Fourth, the JCL comprises a rather large code base, which renders infeasible all purely manual approaches.

There is, however, one common property shared by all methods that implement shortcuts. They all make use of functionality to retrieve information about the current call stack, which is required to be able to check constraints on specific callers on the stack. The task of retrieving this information is typically not delegated to helper methods. By manually reviewing shortcuts we already knew, we found that they either use `sun.reflect.Reflection.getCallerClass` or `java.lang.SecurityManager.getClassContext`. We thus implemented a simple static analysis using the Soot framework [17] to find all methods that contain call sites for these two methods. We only used the Soot framework to locate the specific bytecode instructions conveniently. The analysis does not need a call graph nor does it consider data flows. One could as well have used a text-based matching tool such as *grep*, but using Soot helps avoiding mistakes in the process. Our analysis yielded 86 candidate methods in total, which we reviewed manually to find the subset of methods that actually implement a shortcut. These are the results:

- Out of the 86 candidates, 35 methods do indeed implement a shortcut. They check constraints on the call stack and skip a permission check if these constraints are satisfied.
- Further 6 methods do not implement shortcuts in the strict sense, because they do not call a `check*` method on the `SecurityManager` to trigger stack inspection under any circumstances. Because of this, we consider them to be out of scope. They are noteworthy, however, because they deny access to functionality if the immediate caller's classloader is unable to load a specific class involved in the desired action. Such code implements a kind of undocumented poor man's approach to access control.
- One method does also not implement a shortcut in the strict sense, but it checks if the immediate caller's classloader is the bootstrap classloader, and throws a `SecurityException` otherwise.
- The remaining 44 methods are caller-sensitive, but use stack inspection for purposes other than shortcuts.

We matched our findings with the relevant sections in the JSCG, 9.8, 9.9, 9.10, which provide a list of 75 caller-sensitive methods that have to be used with special care. The methods listed in JSCG constitute a subset of the 86 candidate methods we found by static analysis. The additional 11 methods that we found, which are not covered by the relevant sections in the JSCG, include 9 methods that do not perform permission checking, 1 deprecated method, and 1 method which is part of `sun.misc.Unsafe`, and thus not officially supported. From this, we conclude that the JSCG sufficiently covers the current set of methods that implement shortcuts. For 41 out of the total 75 methods included in the JSCG, we found no indication for shortcuts. In most cases, these methods implement dynamic access checks in the context of reflection, or provide dynamic loading capabilities involving the immediate caller's classloader. Both is caller-sensitive behavior that requires special attention from developers, and might even bring along a potential for vulnerabilities, which is also the reason why they are discussed in the JSCG. We leave these methods out of the scope of this paper, since our focus is on shortcuts, but may be worth investigating further in future work.

B. Removing shortcuts

Out of the 35 methods that we identified to implement a shortcut to bypass proper permission checking, we manually modified 32 of these methods to remove any conditionals that involved properties of the call stack, which may have prevented a permission check from being performed. We found that most shortcuts use `getCallerClass` to retrieve the immediate caller, and check if its defining classloader matches a specific instance, or is `null`, i.e., the bootstrap classloader. By removing shortcuts, we transformed 28 out of these 32 methods from caller-sensitive to caller-insensitive methods, guarding their functionality by a well-defined permission check. We left the remaining 4 methods caller-sensitive after modification because —apart from their original shortcuts— they implement additional functionality, such as visibility checks in the context of reflective access. It is important to stress that caller-sensitivity and the notion of shortcuts, as we defined it, are two separate concepts: Our notion of shortcuts always implies caller-sensitivity but the inverse does not always hold.

As stated above, we removed shortcuts from only 32 out of 35 methods that we found. One of the three remaining methods (`SecurityManager.checkMemberAccess`) we decided to remove entirely from the code base, because it is deprecated and not used by any other method in the JCL.

The other two remaining methods, `Class.getDeclaredField` and `Class.newInstance`, could not be modified, due to circular dependencies. After an initial attempt to modify them, we encountered errors during VM initialization, because using either of the two methods causes a permission check, within which the method itself is called again, which in turn triggers another permission check, and so on. In the original code, the shortcuts in the two methods prevented this call sequence, because, eventually, `newInstance` and `getDeclaredField` would simply skip the permission check and succeed. We did not further investigate whether the use of reflection in the call sequence initiated by a permission check is inevitable. At the same time, we could not come up with a clean solution that would allow the shortcuts to be removed, without making substantial changes to the JCL. We thus decided to keep these two shortcuts and leave all calls to `getDeclaredField` and `newInstance` unmodified.

C. Adapting all callers

The last step of our proposed solution is to adapt all immediate calls to modified methods. In the original code, many of the JCL's callers are able to access functionality guarded by a shortcut, even when there is untrusted code on the call stack, because the shortcut bypasses the permission check. After modification, however, the same call sequence would fail because the permission check now executes, taking into account the full call stack. To retain backward compatibility whenever possible, all immediate calls to modified methods have to be wrapped into a privileged action. As we will explain, the calls to modified methods have to be adapted differently, depending on whether the modified method is still caller-sensitive after modification, or not.

Modifying calls to the 28 methods that lost their caller-sensitivity through modification works as follows. First, we use static analysis to find all immediate calls to any of the modified methods. For this, we adapted and reapplied the approach used to locate shortcuts as described above. As a result, we found 1,399 calls in the JCL that required a modification. For 1 out of the 28 modified methods, we were not able to find even a single caller within the JCL itself, which means that our transformation regarding this method is already complete at this point.

We then used Javassist [18] to implement a bytecode modification tool that automatically adapts all calls. It adds one or more private helper methods to each calling class, each of which instantiates a privileged action that wraps the original target method call and then calls `doPrivileged`. Next, the modification redirects all calls targeting a modified method to one of the newly added helper methods. Note that each helper method wraps a call to one specific modified method only, which is why multiple helper methods are added to calling classes that target more than one modified method. Privileged actions have to be implemented in separate classes (in source code one would normally use an anonymous inner class), but instead of adding one individual implementation for each generated helper method, we added a small set of commonly accessible privileged actions to `java.lang.Class`, shared among all helper methods. By this, we avoid having to add hundreds of additional classes, which would bloat the code base.

We decided to use bytecode modification instead of source-code modification, because at the time we did our experiments, we were not aware of any publicly available source-code modification libraries that would have allowed us to perform the required modifications in an automated fashion.

The calls to one of the 4 methods that remained caller-sensitive after the removal of shortcuts had to be modified differently. This is because those methods vary their behavior depending on the immediate caller, which (as can be seen in "case 2" in Listing 4) would be the `run` method of a privileged action if applying the modification we applied before. In the JCL only `AtomicReferenceFieldUpdater.newUpdater` out of the 4 methods is actually called and this single method has only 3 callers, i.e., we only have to modify 3 callers. It seemed reasonable to modify these 3 callers manually.

```
// case 1: without modification
CallerClass.method // immediate caller
  AtomicReferenceFieldUpdater.newUpdater

// case 2: with "regular" bytecode modification
CallerClass.method
  CallerClass.x_newUpdater
    AccessController.doPrivileged
      PrivilegedActionImpl.run // immediate caller
        AtomicReferenceFieldUpdater.newUpdater

// case 3: alternative modification strategy
CallerClass.method
  CallerClass.x_newUpdater
    AccessController.doPrivileged
      PrivilegedActionImpl.run
        CallerClass.x_getUpdater // immediate caller
          AtomicReferenceFieldUpdater.newUpdater
```

Listing 4. Illustrating the effects the different modification strategies have on the call stack

We applied a modification implementing a form of double dispatch, see "case 3" in Listing 4. First, we manually added two private helper methods, `x_newUpdater` and `x_getUpdater`, to each calling class of `AtomicReferenceFieldUpdater.newUpdater`. `x_newUpdater` instantiates a privileged action, whose `run` method calls `x_getUpdater`. `x_getUpdater`, in turn, calls `AtomicReferenceFieldUpdater.newUpdater`. Finally, we replaced all original calls to `newUpdater` by calls to `x_newUpdater`. The effects of this alternative modification strategy on the call stack can be seen in "case 3" in Listing 4. By routing the call sequence through `x_getUpdater`, instead of immediately calling `newUpdater` in the privileged action, we ensure that the immediate caller of `newUpdater` is the original calling class, not the privileged action. As `newUpdater` is caller-sensitive to the calling class and not the specific calling method it behaves appropriately, i.e., as before.

D. Effects on security and maintainability

The technique presented above removes shortcuts within methods of the JCL. The benefits of these changes are twofold. First, the resulting JCL code is easier to maintain, and in consequence it will be harder to introduce new confused-deputy vulnerabilities in future versions of Java. Second, some of the existing attack vectors that depend on shortcuts will become infeasible.

Enabling security-preserving code evolution

The benefit w.r.t. facilitating security-preserving evolution of the JCL were already highlighted in the background section by discussing the code in Listing 1 and the result of the adaptation by our approach in Listing 2. The desired positive effect of our conversion is that now, if an unprivileged attacker calls `openFileFromRoot`, the permission check will fail, because `Util`, having been added later, was not subject to our modification. This prevents the previous vulnerability.

By trading implicit elevation of privileges with shortcuts for explicit privilege elevation with `doPrivileged` as described above, we retain backward compatibility to a large extent. This has, however, a downside: It will retain confused-deputy vulnerabilities that already existed in the code base at the time of its modification. Consider again the example code in Listing 1. If the vulnerability caused by `Util` is already part of the code base at the time we apply our program transformation, the call to `openFile` in `openFileFromRoot` will be wrapped in a privileged action, just like the call in `readProp`. As a result, the `openFileFromRoot` method will continue to expose critical functionality to attackers even after program transformation.

It is not easy to decide which method calls under privileged regime are legitimate, and which ones represent a vulnerability.

With the current proof-of-concept implementation of our approach, we reduce the possibilities of potentially illegitimate privilege elevation to explicit ones only, but still leave the identification of insecure uses of critical functions out of the scope. In the future, we plan to extend our proposal by a security review of all callers and a decision on a case-by-case basis whether the introduced explicit privilege elevation with `doPrivileged` is appropriate or not. By mapping implicit elevations to explicit ones, the transformation presented in this paper does facilitate such an analysis - as we already argued, the explicit privilege elevations are easier to identify and reason about.

Rendering existing attack vectors infeasible

The good news is that even in its current state of the development, our transformation effectively renders existing attack vectors infeasible. This is, because a large number of attacks that exploited previously shortcut-containing methods did not call these methods directly (like `Util`), but rather by abusing an insecure use of reflection or `invokedynamic` [19]. The proposed transformation does not modify such kinds of calls, as `doPrivileged`-wrappers are only placed around direct method calls. After the shortcuts are removed, any such attack will therefore be successfully thwarted: The permission check in the reflectively called method, from which the shortcut was removed, will now trigger a stack walk, preventing the action if the call sequence was initiated by untrusted code.

As already mentioned, we found four examples of such kinds of attacks in a sample set provided by Security Explorations [12]. They leverage vulnerabilities involving the insecure use of reflection to call shortcut-containing methods through a trusted system class. We verified through debugging that performing permission checks instead of taking the shortcuts will result in access-control exceptions, thus effectively preventing these attacks. Interestingly, after Security Explorations reported three of those vulnerabilities to the vendor, a fix was released that did not reliably prevent the attacks. In fact, it was flawed in many ways, but most importantly, because it still allowed an attacker to make use of shortcuts and other caller-sensitive methods [20]. Consequently, Security Explorations was able to still run three of the four exploits successfully by only changing them slightly.

In conclusion, these findings (a) demonstrate that our proposed solution does increase the security of the Java platform, and (b) also support our claim that shortcut-containing code is very hard to maintain.

V. Performance Evaluation

While our proposed adaptations make the Java platform less vulnerable, will make it easier to maintain and will reduce the chance for security-relevant mistakes while maintaining and extending the platform, the question raises, as what these changes cost in terms of runtime overhead. After all, the shortcuts we remove were originally introduced for the sake of reducing the runtime cost of permission checks [9]. Our evaluation thus addresses the following research question: *Which runtime overhead does the code adaptation introduce?*

A. Evaluation setup

To answer the research question, we transformed OpenJDK 8 b132-03_mar_2014 as described in Section IV and performed several experiments. As baseline we used the same version of the OpenJDK without modifications. To ensure maximum comparability, we built both the modified and the unmodified version ourselves based on the official source release [21].

We compared both variants in two different settings. In the first setting, we run the DaCapo benchmark suite [10] version 9.12-bach on both variants of the Java platform. The goal is to measure the relative overhead that the transformations may induce in the execution of real-world applications. We chose DaCapo because it consists of complex, real-world applications from diverse application domains that cover a broad range of possible program behaviors [10]. Using DaCapo's built-in functionality, we implemented a custom callback class that performs 250 timed runs for each benchmark in each setting, preceded by 750 warm-up runs. We chose such a high number of iterations to minimize the effects of outliers that can be caused by just-in-time compilation or other reasons. By this, we maximize reproducibility of our results and ensure that comparing runtime values is actually meaningful. The following command was used to execute the tests:

```
java -Xcomp -XX:CompileThreshold=1
↪ -server -Xmx2g -Xms2g -Xbatch
↪ -cp ".;./mathlib.jar;./dacapo.jar"
↪ Harness -t 1 -c callback benchmarkname
```

Due to a known bug in DaCapo [22], we had to measure `jython` runtimes without `Xcomp`-flag. We were further required to entirely skip the two benchmarks `batik` and `eclipse` because their execution resulted in errors on both the original and modified OpenJDK. `eclipse` failed during its checksum validation, indicating that the benchmark produced an unexpected output. We were able to reproduce this problem with multiple original Java execution

environments on different machines. We have reported this problem to the benchmark's authors. `batik` fails with a `ClassNotFoundException`, apparently because it accesses a class that is available in Oracle's Java runtime but not in the OpenJDK.

In addition to runtime measurements, we also counted the number of method calls to any of the 32 modified methods that are performed while executing the DaCapo benchmarks. This allows us to reason about the coverage of the DaCapo suite with respect to the proposed changes.

In the second setting, we ran both variants of the OpenJDK on micro benchmarks. These micro benchmarks are artificial test scenarios that we created for all transformed code locations. The goal is to assess the transformed code locations without measuring influences by unaffected code. We do this by calling the first publicly accessible method that transitively calls a modified method. The micro-benchmarking scenario gives an insight into how much the removal of shortcuts costs in the worst case, by calling modified methods frequently. To measure the runtime we used JUnitBenchmarks version 0.72 [23]. We created a dedicated JUnit test case for each modified method, each of which contains minimal setup code, code to prevent dead code elimination [24], and a loop that performs 10,000,000 calls to the method whose runtime is to be measured. This high number of iterations is required because a single invocation is too fast to be measured accurately. JUnitBenchmarks computes the average and standard deviation of the runtime of 10 rounds, and the 10 rounds are preceded by 5 warmup rounds not included in measurements. (A total run for a single test case thus triggers 150,000,000 calls.) The standard deviation is used to gauge the accuracy of the results.

Lastly, we perform both experiments using two different setups. In the first setup we perform the experiments without the presence of a security manager. This setup acts as our baseline. In the second setup, we execute the experiments with a security manager set programmatically, in case of micro benchmarks, and by command line argument, in case of DaCapo (using VM arguments `-Djava.security.manager -Djava.security.policy`). We use the security manager with a policy file granting all permissions to all the code. We manually verified that enabling the security manager actually triggers permission checks performing stack walks at runtime, despite the fact that the code has effectively the same permissions as if no security manager were present.

All experiments were performed on a machine with an Intel Core i5-2400, 3.1Ghz processor, with 4 GB of memory, running a 64-bit Windows 7 Enterprise SP 1.

B. Results on DaCapo

Table I shows the results of the DaCapo benchmark suite. Each benchmark is represented by one row in the table. Column 1 shows the benchmark's name. Columns 2 and 3 show the execution time in seconds without security manager in place, along with the standard deviation for each value. Column 4 shows the runtime difference as a factor to highlight the cost of our proposed solution. Columns 5, 6, and 7 show the same values measured with the security manager in place.

Comparing the runtimes of the original code and the modified code, in almost all cases the difference lies below 1%. In three cases, measurement results indicate that the modified code is faster by 2%. In one exceptional case, the modified code appears to be 3% faster. Taking all results into account, the modified code is at most 1% slower than the original code. We attribute these small runtime differences mostly to instabilities of the JVM [24] rather than to code changes, and can confirm the observation of Gil et al. that even testing identical code may lead to slightly different results in terms of runtime. Furthermore, execution speed is influenced by secondary factors induced by the underlying software/hardware stack, as previously studied by Gu et al. [25]

In addition to runtime measurements, we also collected call statistics[4] to ensure that the modified methods are actually involved in benchmark execution. Our results clearly show that this is the case. Table II shows a summary of the results we measured for the original OpenJDK without a security manager. Each of the twelve DaCapo benchmarks is represented in one row. Column 1 shows the benchmark's name, column 2 shows the total number of method calls to any of the modified methods, column 3 shows the number of modified methods the benchmark uses. Finally, column 4 shows the modified method that was most frequently used by the respective benchmark. As can be seen, the DaCapo benchmark suite extensively uses most of the 32 methods under investigation. Running any of the benchmarks requires at least 11 out of 32 methods, and 22 at most. Only eight out of 32 modified methods are not used at all by DaCapo. Further, executing just a single run of one of the benchmarks involves between 147 and 668,000 calls to modified methods. A single run of the entire benchmark suite requires more than 900,000 calls to the modified methods.

Summarizing the results, we conclude that the proposed code changes have virtually no performance impact on the tested real-world applications, even though the modified methods are heavily used. At a first glance, this result seemed surprising even to us, which is why we sought to confirm it through micro-benchmarks...

C. Results on micro benchmarks

We implemented 32 tests using JUnitBenchmarks, which equals the total number of shortcut methods found (35), excluding `newInstance`, `getDeclaredField`, and `checkMemberAccess`. The former two methods we could not modify, the latter we removed during program transformation (see Section IV). Each test performs 10,000,000 calls to the method under investigation, in two exceptional cases, due to long runtimes, we performed only 200,000 calls and interpolated the results.

Table III shows a summary of the results of the micro benchmarks. To avoid misunderstandings in the following discussion, all results are shown in microseconds per single

[4]Complete call statistics are provided with the artifacts.

TABLE I
RUNTIMES OF DACAPO IN SECONDS

Project	Without `SecurityManager`				With `SecurityManager`			
	Original	Modified	Overhead abs.	Overhead rel.	Original	Modified	Overhead abs.	Overhead rel.
avrora	3.08 ±[0.15]	3.11 ±[0.10]	0.03	1%	3.02 ±[0.06]	3.06 ±[0.02]	0.04	1%
fop	0.31 ±[0.01]	0.31 ±[0.01]	0.00	1%	0.32 ±[0.01]	0.32 ±[0.01]	0.00	1%
h2	3.70 ±[0.01]	3.70 ±[0.01]	0.00	0%	3.67 ±[0.01]	3.70 ±[0.01]	0.00	1%
jython	1.48 ±[0.02]	1.47 ±[0.02]	-0.01	-1%	1.50 ±[0.02]	1.47 ±[0.03]	-0.03	-2%
luindex	1.02 ±[0.12]	0.99 ±[0.05]	-0.03	-2%	1.20 ±[0.07]	1.21 ±[0.08]	0.01	1%
lusearch	4.95 ±[0.02]	4.95 ±[0.02]	0.00	0%	4.99 ±[0.01]	4.86 ±[0.02]	-0.13	-3%
pmd	2.50 ±[0.02]	2.48 ±[0.02]	-0.02	-1%	3.06 ±[0.03]	3.05 ±[0.04]	-0.01	0%
sunflow	8.36 ±[0.03]	8.31 ±[0.02]	-0.05	-1%	8.36 ±[0.04]	8.34 ±[0.03]	-0.02	0%
tomcat	48.54 ±[0.28]	48.56 ±[0.31]	0.02	0%	52.54 ±[0.81]	52.35 ±[0.35]	-0.19	0%
tradebeans	8.71 ±[0.02]	8.83 ±[0.03]	0.12	1%	10.01 ±[0.05]	10.02 ±[0.02]	0.01	0%
tradesoap	17.94 ±[1.27]	17.67 ±[0.89]	-0.27	-2%	23.22 ±[1.44]	23.54 ±[2.02]	0.32	1%
xalan	6.54 ±[0.04]	6.60 ±[0.04]	0.06	1%	6.76 ±[0.03]	6.79 ±[0.02]	0.03	1%
				∅ -0.25%				∅ 0.08%

TABLE II
CALL STATISTICS FOR DACAPO

Project	Calls	Methods	Most freq. used
avrora	147	11	getClassLoader
fop	6,329	11	getContext-ClassLoader
h2	210	11	getClassLoader
jython	1,483	19	getMethod
luindex	208	11	getClassLoader
lusearch	159	11	getClassLoader
pmd	1,885	12	getClassLoader
sunflow	249	12	getClassLoader
tomcat	32,069	17	getDeclared-Methods
tradebeans	219,792	22	getContext-ClassLoader
tradesoap	668,000	22	getFields
xalan	36,696	11	getParent
	∅ 80,602	∅ 14	
	Σ 967,227	∪ 24	

TABLE III
SUMMARY OF RUNTIMES OF MICRO BENCHMARKS IN MICROSECONDS (µS) PER SINGLE INVOCATION

	Without `SM`			With `SM`		
	Orig.	Mod.	Ovh.	Orig.	Mod.	Ovh.
Min.	<0.01	<0.01	-0.27	0.03	0.58	0.22
Max.	31.70	49.25	17.55	79.15	99.65	20.50
Avrg.	1.94	2.94	1.00	3.80	6.18	2.38
Med.	0.29	0.16	0.00	0.45	2.05	1.17

invocation, instead of seconds per 10,000,000 calls. The first column shows the minimum, maximum, average, and median runtimes of the original OpenJDK without a security manager in place. The second column shows the respective results for the modified OpenJDK. Column 3 shows minimum, maximum, average, and median values of the absolute runtime differences between the original and modified code. Columns 4, 5, and 6 show the respective results for the tests we performed with a security manager in place.

Our first observation is that all methods under investigation execute extremely fast, and that is before and after modification. As can be seen in columns 1 and 2, without a security manager in place, a single call to the fastest method completes in <0.01 µs, both on the original and modified OpenJDK. The slowest method requires 31.70 µs on the original code, and 49.25 µs on the modified code, which equals an overhead of 17.55 µs.

For the tests without the security manager, this is the highest absolute impact that we encountered. The result set contains one more outlier with an overhead of 12.7 µs, while the remaining 30 methods show differences in runtime between -0.27 µs and 1.82 µs. In fact, in this setting only 13 out of 32 methods show a performance penalty at all. Based on the call statistics we collected for DaCapo, we can say that one of the two outliers is not used at all, while the other one is used more than 16,000 times in the entire suite.

On average, without the security manager, the original OpenJDK executes the methods under investigation in 1.94 µs per single call. The modified OpenJDK takes 2.94 µs, which is an average overhead of 1 µs per method call induced by our proposed transformations. It is important to note, however, that the two outlier methods mentioned before influence average values to a greater extent than all the other methods. The median execution time for a single method call without security manager on the original OpenJDK is 0.29 µs, compared to 0.16 µs on the modified OpenJDK, demonstrating that several methods even became faster. Overall, the runtimes we measured without the security manager in place show that, if at all, there are only insignificant performance penalties induced by our proposed code changes.

The performance measurement results for tests performed with the security manager in place are not much different to the results without the security manager. As can be seen in columns 4 and 5, the minimum execution time increased from 0.03 µs per single method call to 0.58 µs. The method that originally completed in 0.03 µs shows the largest relative overhead induced by our code transformations. It needs 2 µs after modification and is thus not the fastest method in the

modified OpenJDK anymore. It is one of four outliers with a relative overhead of >1000%: `getParent`, `getContextClassLoader`, `getClassLoader`, `getSystemClassLoader`. In all these cases, however, the original execution time was significantly <0.1 µs per single method call, and <2.1 µs after modification, which we still consider extremely fast. As can be seen in Table II, three out of those four outliers are the most frequently used modified method by at least one of the benchmarks. We can thus say that DaCapo provides good coverage in that respect.

In terms of absolute overhead, the result set includes two outliers with an overhead >10 µs per single call, while the remaining 30 methods show overheads between 0.2 µs and 4 µs. Those two outliers are the same methods that showed the largest absolute overhead without the security manager. One of those two outliers is the longest running method before and after transformation. As can be seen in the second row in columns 4 and 5, it has a runtime of 79.15 µs per single call in the original OpenJDK, and 99.65 µs in the modified OpenJDK. These two outliers, as discussed before, greatly influence the average runtimes. With the security manager in place, the average runtime increases from 3.80 µs per single method call to 6.18 µs, which is an average overhead of 2.38 µs. The median runtime increases from 0.45 µs to 2.05 µs per single method call, and the median overhead is 1.17 µs. None of the methods under investigation became faster through our modifications, if the security manager is in place.

In summary, one can see that when measuring modified methods in isolation there is a measurable performance penalty. However, these penalties are very small in absolute terms, which is why they do not influence the runtimes of real-world applications such as the DaCapo benchmarks.

D. Reason for lack of performance effects

The positive performance results might appear surprising at first, however, there is a simple explanation for why a shortcut-free implementation does not suffer from performance penalties. The reason is that the calls to `doPrivileged` that the proposed hardening introduces have, in terms of performance, a similar effect to shortcuts: At runtime, they cause the stack walk to terminate early. The JVM checks only the permissions of code that executes within the `doPrivileged`-wrapper but not the calling code's permissions. This greatly reduces the number of stack frames that permission checks must traverse and avoids a performance penalty with current JVM technology.

VI. Productive Use and Further Research

Our proposed solution is functional and comprehensive, because it allows for the execution of legacy applications and it avoids the dangers of implicit privilege elevation. Our proof of concept code shows that such a change is possible without significantly impacting the runtimes of a set of real-world applications. However, implementing our proposed solution for productive use requires reconsideration of two aspects: (a) policies for legacy applications may have to be adjusted when switching from a shortcut-containing platform to a shortcut-free platform, and (b) standard permissions of the Java platform cannot equivalently represent some of the privileges originally gained through shortcuts in terms of their semantics. In the following, we will elaborate on these issues and discuss the solution space to spark further research and to aid an actual application of the proposed hardening into Java's code base. We have reported our findings to the security team at Oracle Inc. who is, based on our discussions here, considering an application of this hardening for a future version of Java.

Adjusting security policies

An application's privileges are usually defined by a set of permissions granted explicitly in a security policy. This is not the case for privileges gained through shortcuts, because they are hard-wired into the JCL. Removing shortcuts will cause permission checks to be executed that would have been skipped otherwise, which will require permissions that were not needed before the change. Some legacy applications will thus require adjustments to their security policy when upgrading to a runtime environment that is shortcut-free. This task can either be done manually by determining the required permissions through code reviews and dynamic testing, or automatically by means of a static analysis that computes the set of required permissions for any given application class. Appropriate approaches have been proposed earlier [26].

Not all legacy applications are affected by this issue. No changes of the security policy are required for applications that do not immediately call shortcut-containing methods, call them in a way that does not trigger a shortcut, have already been granted all required permissions anyway, or run without a security manager.

Reworking Java's standard permissions

Java's standard permissions cannot equivalently express all privileges gained through shortcuts in terms of their semantics. As an example, a shortcut in `Class.getDeclaredFields` skips a permission check if callers attempt to access fields of classes that were loaded with the same classloader. After removing this shortcut, the permission check will always be executed and all callers will be required to have permission *RuntimePermission 'accessDeclaredMembers'*. As explained before, the security policies of legacy applications can be updated to grant this permission when upgrading to a shortcut-free runtime environment with little to no effort. However, the problem is that granting this permission provides applications more privileges than the original shortcut implementation, as it will allow callers to access fields of *arbitrary* classes, including private members of system classes. Note that the security implications of this are very limited because *RuntimePermission 'accessDeclaredMembers'* only allows for member access, i.e., retrieving instances of `java.lang.reflect.Field` or `java.lang.reflect.Method`, and not for reading or writing any private field values, or calling private methods.

This is nevertheless an issue, and it is caused by the fact that some standard permissions are too coarse-grained to be used in a meaningful manner. Permissions that are supposed to restrict the use of reflection only allow for on/off decisions, thus either alowing reflective access to all available classes, or none at all.

The consequence of coarse-grained standard permissions is that, when upgrading to a shortcut-free platform, applications may have to be granted permissions that provide more privileges than originally granted through shortcuts. This is inherently risky, as it violates the principle of least privilege. Even when letting the consequences of our proposed changes aside, we consider permissions like the reflection permissions as too coarse to be really useful in a security-sensitive setting.

This circumstance is not caused by technical issues, but is simply a design flaw. Consider Java's standard file permission, which, in contrast to the reflection permissions, provides great flexibility for fine-grained access decisions. It allows for specifying the file path to which the permission applies, as well as the specific actions that shall be granted, such as 'read' or 'write'. Similar expressiveness is desired for securely restricting the use of reflection of untrusted applications, and for adequately compensating for the removal of shortcuts. We thus argue that a thorough redesign of Java's standard permissions is both possible and required. This is a complex task in itself that needs to take into account technical aspects, as well as various organizational and human factors. One of the major challenges is to allow for fine-grained access-control decisions that support the principle of least privilege, without being hard to use or performance-wasting. Further, the permission model should be designed to better support automatic policy generation for existing applications. We hope that future research will take on the challenge of developing a permission model that is both flexible, and usable within the settings that it is designed for.

VII. Lessons Learned

The work presented here specifically focuses on access control as it is implemented in Java. Due to the widespread use of this platform, our results are of high relevance to the security of a large number of servers and workstations. Furthermore, the Java security model is an interesting research subject, as it is one of the most sophisticated models of its kind found in modern software. The rigorous analysis presented here sheds light on how such a complex model is weakened in practice. Besides the impact our results may have on further developments of the Java platform, we can also view this work as a case study and derive a set of general recommendations for the development of secure software. In the following, we will highlight general lessons learned, that hopefully serve as guidance for the design and implementation of other complex security models.

Explicit privilege elevation aids the protection of privileges.

Our research clearly shows that by elevating privileges explicitly through constructs such as `doPrivileged`, one can avoid the accidental reexposure of those privileges to attackers. One reason is that `doPrivileged` elevates privileges temporarily and only within a given lexical scope. Any code refactorings performed will move the explicit `doPrivileged`-call along with the other code, causing privileges to be raised only where required. A second reason, though, is the pure presence of the `doPrivileged`-call. To JCL maintainers it not only serves as a security construct but can serve also as a warning flag: privileges are elevated at this point and need to be properly protected from being leaked to the outside.

Stick to the security model.

Security models of complex systems are planned and designed prior to implementation. Inconsistencies between design and implementation can be risky as they hamper proper evaluation and maintenance. In the concrete case we studied here, shortcuts are used instead of proper stack-based access control, which is a deviation from the Java security model that increases the attack surface. A common practice in software engineering is to readjust a project plan if it drifts apart from reality. It should be just as normal to readjust and revaluate a security model if strictly implementing it as prescribed is not possible, e.g., due to performance constraints. In the specific case of Java, had our evaluation been performed earlier, one would probably have had the chance to design a more fine grained policy system in the first place, which then in turn would have allowed all current use cases without having to opt for implicit privilege elevation.

Properly document tradeoffs between security and performance.

Design and implementation of software is shaped by functional and non-functional requirements. Tradeoffs are often necessary due to conflicting requirements, and security-related functionality not always has the highest priority. While in many cases at least functional requirements are documented, it seems less common to properly document how tradeoffs shaped the design and implementation of a complex software. In the specific case of shortcuts in Java's access control mechanisms, we were required to perform manual reviews, functional tests, and also doublecheck with representatives from Oracle to verify our assumption that one reason for which shortcuts were introduced was for performance reasons. In result, performance-related tradeoffs in long-living systems should be thoroughly documented, as performance constraints definitely change over time and many optimizations become obsolete eventually.

Revaluate performance tradeoffs in regular intervals.

The very nature of a tradeoff is to balance out a negative impact with a positive impact of similar or higher value. If the hoped-for positive impact is improved performance and ease of use, and the negative impacts are, e.g., an increased attack surface and decreased maintainability, then this tradeoff changes as performance gains decrease with optimizations of

the runtime. We thus argue that a revaluation of performance-related tradeoffs in regular intervals should be part of the maintenance process of any long-living system.

VIII. Related Work

Aside from Li Gong's extensive work [2], [15], [27] for the Java security model, several researchers took up the challenge to analyze, extend or break the model. We present work that is concerned with optimizing the access-control process as well as work that goes beyond this and seeks out for alternatives to stack-based access control.

Fournet and Gordon [28] provide a complete theoretic model for stack-based access control. Using this model they are also able to point out the limitations of stack inspection. Shortcuts, on the other hand, are not part of the model and may invalidate the guarantees achieved with it.

Herzog et al. [29] analyzed the performance of the `SecurityManager` in Java and provide guidance on how to use it efficiently. However, they do not provide any details on shortcuts.

Several approaches have been developed to optimize Java stack inspection in order to reduce performance overheads. Bartoletti et al. [7] present two control-flow analyses that safely approximate the set of permissions granted or denied to methods and therefore speed up runtime checks. Koved et al. [26] extend the precision and applicability of the analysis. Chang [30] built on their work to make the analyses more precise using a backward static analysis to compute more precise information on the performed checks. Likewise, Pistoia et al. [8] analyze Java bytecode to find unnecessary (and therefore excessive) and redundant (and therefore inefficient) privileges while ensuring that there are no tainted variables in the privileged code. This is not only interesting for optimization but also helpful in the detection of vulnerabilities.

Other work is more concerned with the maintainance issue of access control in Java. Cifuentes et al. [9] provide a definition for caller sensitivity and describe means to detect unguarded caller sensitive method calls. Toledo et al. [31] observed that access-control checks are scattered throughout the JCL making them non-modular and therefore hard to maintain. They propose two solutions based on aspect-oriented programming to fully modularize access control in Java. In their work they not only cover permission checking, but also privileged execution and permission contexts.

Moreover, there are alternatives to stack-based access control. Abadi et al. [32] suggest to base access control on execution history rather than on the current call stack. This will not only capture the nesting of methods, but also any method that has completed prior to the method that is checked. Methods that already completed are not on the call stack anymore and would thus be ignored by regular stack-based access control. Nevertheless, such methods may change the global state of the application to a state in which subsequently called sensitive methods should not be allowed to execute. Martinelli et al. [33] integrated history-based access control into Java using IBM's Jikes VM. However, this approach causes a significant slowdown as its checks are more costly.

Wallach et al. [34] discuss an alternative they call *security-passing style*. They represent security contexts as pushdown automata, where calling a method is represented by a push operation and returning is represented by a pop operation. To weave these automata with the program they rewrite a program's bytecode so that it no longer needs any security functionality from the JVM.

Building on the work on stack-based and history-based access control, Pistoia et al. [35] introduce information-based access control. They argue that history-based access control may prevent authorized code from executing because of less authorized code executed previously, although it may not have influenced the security of the operation that is about to be executed. In information-based access control every access-control policy implies an information-flow policy. It augments stack inspections with the tracking of information flow to sensitive operations. An extensive review on the relation of access control and secure information flow is given by Banerjee and Naumann [36].

IX. Conclusion

A key contribution of this paper is the thorough analysis of the security threat imposed by current shortcuts in the Java Class Library (JCL), which omit stack-based access-control in certain situations and cause implicit privilege elevation. The presence of shortcuts is responsible for the single largest group of vulnerabilities known to have been exploited for the Java runtime. We showed that shortcuts directly enable attack vectors and complicate the security-preserving maintenance and evolution of the code base; as they elevate privileges to certain callers implicitly, their callers are in many cases either unaware of the elevations or unable to reason about their effects and scope.

Through a tool-assisted adaptation we have created a new variant of the JCL that works almost without shortcuts, allowing privileges to be elevated only explicitly through the use of privileged wrappers. The adapted code allows maintainers, security experts and tools to easily identify points of privilege escalation. Moreover, some previous exploits that abuse insecure use of reflection are effectively mitigated by our approach.

One reason for which shortcuts were originally introduced was to lower the execution overhead of access control. Surprisingly at first, however, a set of large-scale experiments with the DaCapo benchmark suite shows virtually no measurable runtime overhead caused by our removal of shortcuts. Micro-benchmarks explain this result by showing that, in the worst case, the absolute overheads introduced are all in the order of microseconds. As we discussed, the reason for this positive performance is due to early stack-walk terminations at `doPriviliged`-calls.

A second reason for the presence of shortcuts is that the implicit assignment of privilege is convenient, as it reduces

the need to elevate priviliges explicitly, e.g. through an appropriate access-control policy. We thus assessed in detail the usability implications that a move from implicit to purely explicit privilege elevation entails. The tradeoffs discussed will ultimately determine whether the proposed hardening is worthwhile adopting at this point in time.

Another major point of consideration for adopting the proposed hardening is the large one-time cost involved in implementing it: ideally, security-trained JCL developers should review every single `doPrivileged`-call our adaptation introduces, to see that it is not unduly leaking privilege. We have reported our findings to the security team at Oracle Inc. and are discussing those tradeoffs with them. In future work, we plan to work towards tool support for proving privilege containment at least for some recurring situations.

REFERENCES

[1] "About Java," https://www.java.com/en/about/.

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

[3] "2013 cisco annual security report," http://www.cisco.com/web/offer/gist_ty2_asset/Cisco_2013_ASR.pdf, 2013.

[4] "2014 cisco annual security report," http://www.cisco.com/web/offers/lp/2014-annual-security-report/index.html, Jan. 2014.

[5] P. Holzinger, S. Triller, A. Bartel, and E. Bodden, "An in-depth study of more than ten years of java exploitation," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 779–790.

[6] N. Hardy, "The confused deputy:(or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[7] M. Bartoletti, P. Degano, and G. Ferrari, "Static analysis for stack inspection," *Electronic Notes in Theoretical Computer Science*, vol. 54, no. 0, pp. 69 – 80, 2001, conCoord: International Workshop on Concurrency and Coordination (Workshop associated to the 13th Lipari School). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066104002361

[8] M. Pistoia, R. Flynn, L. Koved, and V. Sreedhar, "Interprocedural analysis for privileged code placement and tainted variable detection," in *ECOOP 2005 - Object-Oriented Programming*, ser. Lecture Notes in Computer Science, A. Black, Ed. Springer Berlin Heidelberg, 2005, vol. 3586, pp. 362–386. [Online]. Available: http://dx.doi.org/10.1007/11531142_16

[9] C. Cifuentes, A. Gross, and N. Keynes, "Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform," in *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2015. New York, NY, USA: ACM, 2015, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/2771284.2771286

[10] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, 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 *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 169–190. [Online]. Available: http://doi.acm.org/10.1145/1167473.1167488

[11] "Secure coding guidelines for Java SE," http://www.oracle.com/technetwork/java/seccodeguide-139067.html.

[12] "Security Explorations - SE-2012-01 Details," http://www.security-explorations.com/en/SE-2012-01-details.html.

[13] K.-P. Yee, "User interaction design for secure systems," Computer Science Division (EECS), University of California, Tech. Rep., 2002.

[14] S. Türpe, "Idea: Usable platforms for secure programming – mining unix for insight and guidelines," in *Engineering Secure Software and Systems (Proc. ESSoS'16)*, ser. LNCS, vol. 9639, Apr. 2016, forthcoming. [Online]. Available: http://testlab.sit.fraunhofer.de/downloads/Publications/tuerpe2016idea.pdf

[15] L. Gong, "Java security: A ten year retrospective," in *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, Dec 2009, pp. 395–405.

[16] "National vulnerability database," https://nvd.nist.gov/.

[17] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in *Cetus Users and Compiler Infrastructure Workshop*, 2011. [Online]. Available: http://www.bodden.de/pubs/lblh11soot.pdf

[18] S. Chiba, "Javassist-a reflection-based programming wizard for java," in *Proceedings of OOPSLA98 Workshop on Reflective Programming in C++ and Java*, 1998, p. 174.

[19] J. R. Rose, "Bytecodes meet combinators: Invokedynamic on the jvm," in *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL '09. New York, NY, USA: ACM, 2009, pp. 2:1–2:11. [Online]. Available: http://doi.acm.org/10.1145/1711506.1711508

[20] "Security Explorations - Security vulnerability notice," http://www.security-explorations.com/materials/SE-2012-01-IBM-2.pdf.

[21] "OpenJDK source releases - build b132," http://download.java.net/openjdk/jdk8/.

[22] "The dacapo benchmark suite - #80 jython generates npe with eager compilation," https://sourceforge.net/p/dacapobench/bugs/80/.

[23] "JUnit Benchmarks," http://labs.carrotsearch.com/junit-benchmarks.html.

[24] J. Y. Gil, K. Lenz, and Y. Shimron, "A microbenchmark case study and lessons learned," in *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*. ACM, 2011, pp. 297–308.

[25] D. Gu, C. Verbrugge, and E. M. Gagnon, "Relative factors in performance analysis of java virtual machines," in *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 2006, pp. 111–121.

[26] L. Koved, M. Pistoia, and A. Kershenbaum, "Access rights analysis for java," in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '02. New York, NY, USA: ACM, 2002, pp. 359–372. [Online]. Available: http://doi.acm.org/10.1145/582419.582452

[27] L. Gong, "Secure java class loading," *Internet Computing, IEEE*, vol. 2, no. 6, pp. 56–61, Nov 1998.

[28] C. Fournet and A. D. Gordon, "Stack inspection: Theory and variants," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 3, pp. 360–399, May 2003. [Online]. Available: http://doi.acm.org/10.1145/641909.641912

[29] A. Herzog and N. Shahmehri, "Performance of the java security manager," *Computers & Security*, vol. 24, no. 3, pp. 192–207, 2005. [Online]. Available: http://dx.doi.org/10.1016/j.cose.2004.08.006

[30] B.-M. Chang, "Static check analysis for java stack inspection," *SIGPLAN Not.*, vol. 41, no. 3, pp. 40–48, Mar. 2006. [Online]. Available: http://doi.acm.org/10.1145/1140543.1140550

[31] R. Toledo, A. Nunez, E. Tanter, and S. Katz, "Aspectizing java access control," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 101–117, Jan 2012.

[32] M. Abadi and C. Fournet, "Access control based on execution history," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/7.pdf

[33] F. Martinelli and P. Mori, "Enhancing java security with history based access control," in *Foundations of Security Analysis and Design IV*, ser. Lecture Notes in Computer Science, A. Aldini and R. Gorrieri, Eds. Springer Berlin Heidelberg, 2007, vol. 4677, pp. 135–159. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74810-6_5

[34] D. S. Wallach, A. W. Appel, and E. W. Felten, "Safkasi: A security mechanism for language-based systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, pp. 341–378, Oct. 2000. [Online]. Available: http://doi.acm.org/10.1145/363516.363520

[35] M. Pistoia, A. Banerjee, and D. Naumann, "Beyond stack inspection: A unified access-control and information-flow security model," in *Security and Privacy, 2007. SP '07. IEEE Symposium on*, May 2007, pp. 149–163.

[36] A. Banerjee and D. A. Naumann, "Stack-based access control and secure information flow," *J. Funct. Program.*, vol. 15, no. 2, pp. 131–177, 2005. [Online]. Available: http://dx.doi.org/10.1017/S0956796804005453