A Systematic Hardening of Java's Information Hiding

Philipp Holzinger
Fraunhofer SIT
Darmstadt, Germany
philipp.holzinger@sit.fraunhofer.de

Eric Bodden
Paderborn University & Fraunhofer IEM
Paderborn, Germany
eric.bodden@uni-paderborn.de

ABSTRACT

The Java runtime is installed on billions of devices worldwide, and over years it has been a primary attack vector for online criminals. In this work, we address that many attack vectors exploit weaknesses in Java's information hiding, making use of illegal access to private members of system classes. To study to what extent such attacks can be mitigated, and at what cost, this paper demonstrates a proof-of-concept solution to strengthen information hiding. Experiments show that this approach is backward compatible, and that it blocks 84% of all information-hiding attacks in a large-scale sample set at an average performance overhead below 2%. Based on our experiments, we suggest a solution to strengthen information hiding for productive use that has the potential to outperform our proof of concept in terms of robustness and performance, and also would block the remaining information-hiding attacks. Finally, we conclude with general advice on the design of secure software.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Access control**; Browser security.

KEYWORDS

Java, information hiding, access control, security design, exploit

ACM Reference Format:
Philipp Holzinger and Eric Bodden. 2021. A Systematic Hardening of Java's Information Hiding. In *Proceedings of the 2021 International Symposium on Advanced Security on Software and Systems (ASSS '21), June 7, 2021, Virtual Event, Hong Kong*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3457340.3458300

1 INTRODUCTION

Since its introduction in the mid-90's, the Java platform has seen widespread adoption, and is now present on literally billions of devices worldwide. It is one of the first runtime environments that was originally designed to securely contain the execution of untrusted code, for which it implements an elaborate security model [17]. Yet, as Cisco's Annual Security Reports show, Java was the primary attack vector for web exploits in 2013 with a share of 87% [9], and even 91% in 2014, thus clearly outranking Flash and Adobe PDF [10], and still remains an important target today.

At a first glance this seems rather curious, given that Java implements numerous defense strategies: Java's bytecode validation, for instance, statically type-checks all bytecode prior to its execution. Dynamic policy enforcement uses stack-based access control to restrict which sensitive operations untrusted code may execute, while automatic memory management seeks to prevent memory-corruption vulnerabilities. Altogether Java's security architecture involves more than a dozen different safeguards, which might lead one to believe that every mechanism is backed up by at least one other mechanism, thus implementing *defense in depth.* In this work we show, however, that in practice all those mechanisms provide largely orthogonal guarantees. Because of this, a single failure in any one mechanism can often enable attackers to *completely bypass all security checks.* As we explain, the problem is caused by the fact that the Java runtime's security relies heavily on the principle of information hiding: The security status of the entire platform depends on a small set of private variables in system classes. By obtaining access to those variables, or to certain private system methods, attackers can disable all security checks. Using a set of exploits covering more than a decade of Java exploitation [21], we show that 61% of the exploits attack information hiding.

In recent years, Java's maintainers have fixed numerous individual vulnerabilities that had enabled those exploits. As we show in this paper, however, the vulnerabilities are just symptoms of a major underlying design weakness.

This paper gives a detailed account of *two* possibilities to strengthen information hiding such as to avert all attacks known to date. The first, the *heavyweight* mitigation strategy is clean and simple to explain, yet requires deep and extensive modifications of the Java virtual machine and its inner workings. Such modifications would require access to the Java virtual machine's source code, and significant engineering efforts to integrate our solution into Java runtimes of different vendors, on different platforms, in different versions, to support large-scale testing. This heavyweight solution is very useful for future platform implementation, but unfortunately precludes us from implementing and assessing this solution for the closed-source Java runtimes provided by Oracle and IBM.

Hence, to study the extent to which such attacks can be mitigated by a systematic solution that does not require modifications of the Java virtual machine, we also present a *lightweight* proof-of-concept approach that strengthens Java's information hiding by moving sensitive fields into a securely-encapsulated storage area, the "blackbox". This automatically renders impossible all accesses to those fields through previously exploited channels, such as reflection. The hardening further comprises a dedicated protection of private methods, classpool layout randomization, and integrity checks. Our implementation automatically integrates the proposed

ASSS '21, June 7, 2021, Virtual Event, Hong Kong
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8403-2/21/06…$15.00
https://doi.org/10.1145/3457340.3458300

changes into past and current versions of the Java platform, without requiring access to its source code.

Our evaluation shows that this proof-of-concept hardening successfully blocks 84% of all information-hiding exploits in a large-scale sample set, and we explain how even the remaining exploits can be blocked as well. At the same time, the approach is backward compatible and highly efficient—all real-world applications we tested run unaffectedly, with an average performance overhead below 2%.

As we also explain, if implemented, the *heavyweight* mitigation strategy has the potential to outperform the lightweight proof-of-concept approach in terms of both speed and robustness.

This paper presents the following core contributions:

- A systematic analysis of Java's information hiding, which highlights systematic flaws in Java's security model.
- A systematic hardening of Java's information hiding as a proof-of-concept implementation.
- An analysis showing that the proof of concept blocks 84% of past information hiding exploits, while retaining backward compatibility and high performance.
- An elaborate solution approach to strengthen information hiding for productive use, which improves over the proof of concept.
- A general discussion on secure software design, taking into account the insights we gained during our study.

This paper is organized as follows. Section 2 elaborates on the importance of information hiding in the Java security model. Section 3 discusses threats to information hiding, and Section 4 presents our lightweight proof-of-concept solution to strengthen the platform. Section 5 evaluates the proposed lightweight countermeasures. Section 6 discusses the heavyweight redesign of Java's security architecture for productive use, and in Section 7 we draw general conclusions on the design of secure software. Finally, Section 8 presents related work, and Section 9 concludes.

2 BACKGROUND

At a high level, the platform comprises two components. The Java virtual machine (JVM) is the program execution engine. It features a classloader, bytecode interpreter, just-in-time compiler, and other basic functionality. The second component is the Java class library (JCL), which interfaces between the user-provided application and the JVM. It consists of numerous trusted system classes, which offer commonly-needed functionality to applications, similar to the standard libraries of other languages. While the JVM is implemented in native code, most parts of the JCL are implemented in Java. The Java native interface (JNI) can be used to call native functions from Java methods, and vice versa.

The platform enforces policies to restrict access to security-critical functionality (e.g., file and network access) in the JCL by combining several security mechanisms. Some of these mechanisms are implemented in the JVM, while others are implemented in the JCL itself. Security-sensitive methods in the JCL are guarded by a permission check, which is triggered by a call to the `SecurityManager`. A permission check inspects the call stack to determine if the sequence of callers that attempt the desired action are associated with sufficient privileges, and throws a `SecurityException` otherwise. This stack-based access control is one of Java's core security mechanisms. It can be enabled and disabled by the user, but for the remainder of this paper we assume that a security manager has been put in charge. Besides, there is also, e.g., bytecode verification, which ensures that class files are well-formed, type-safe and contain only legal operations, or automatic memory management, which is responsible for the automatic deallocation of memory, thus avoiding memory-corruption vulnerabilities.

Information hiding is crucial in policy enforcement. The security status of the entire platform is determined by the contents of private variables in system classes. One prominent example of such a security-critical field is `System.security`. It holds the instance of `SecurityManager` that is currently in charge of performing permission checks. Setting this field to `null` will disable policy enforcement, thus enabling arbitrary code execution. Another example can be found in `java.beans.Statement`. Objects of this class represent arbitrary method calls. They can be instantiated during runtime and executed by calling `execute()`. To prevent attackers from gaining privileges by calling a sensitive method through `Statement` as opposed to calling it directly, `Statement` captures the current security context in a private field `acc` and ensures that `execute()` will call the target method only within this context. However, attackers can elevate their privileges by illegitimately replacing the security context stored in `acc` by another context with all privileges, as demonstrated by the exploit for CVE-2013-2465.

Finally, system classes store instances of *restricted classes* in private variables. Restricted classes are a subset of system classes that must remain inaccessible to untrusted code, because they contain security-sensitive methods that are not guarded by permission checks. A well-known example of a restricted class is `sun.misc.Unsafe`, which many exploits abuse to disable security checks.

3 THREAT MODEL

In the following we outline our assumptions with respect to attackers and attacks that shaped the design of our solution.

3.1 Attacker goals and capabilities

In our threat model, the attacker's primary goal is to escalate privileges of attacker-controlled code executed by a JRE on the target machine—with security manager enabled. For this, the attacker breaks information hiding by exploiting security vulnerabilities in the JRE to gain access to sensitive private class members in system classes, which in turn are used to achieve privilege escalation. Eventually, a full compromise of the JVM becomes possible.

For the defenses we present in this paper, we assume that the attacker has the following capabilities. The presented defenses are designed to withstand attackers with those capabilities.

- Execute Java code with a restricted set of privileges on the target machine.
- Exploit vulnerabilities in the Java runtime that allow for illegitimate access to private sensitive variables and methods in system classes.
- Statically and dynamically analyze the installation files of the Java runtime as they are deployed by the vendor.

We further assume the following restrictions.

- The attacker cannot intercept method calls between system classes, their call arguments are confidential. This assumption is also relevant for the original Java security model.
- The attacker cannot modify the Java runtime on the target machine in order to prepare the attack. The binary representations of all system classes and native components of the Java platform on the target machine are out of reach for the attacker. This assumption is certainly realistic, unless the target has been compromised already through other means.

3.2 Attack vectors to break information hiding

We reviewed a set of 61 exploits that Holzinger et al. already studied in depth in previous work [21]. This set is the largest sample set of Java exploits available, covering more than a decade of Java insecurity. We found that 37 of the exploits directly break Java's information hiding, for which they are using one of the following three ways. Interestingly, each of these attack vectors can be implemented by exploiting just a single security vulnerability, which supports our claim that the Java security model systematically lacks defense in depth.

Insecure use of introspection – The reflection API and `MethodHandles` allow for class introspection at runtime. Their use is restricted and hence subject to policy enforcement, which is implemented through stack-based access control—relevant methods in, e.g., the reflection API are guarded by a call to the security manager, which ensures that all code involved in the action has been assigned appropriate privileges. However, system classes can use introspection APIs in an insecure manner, thus turning them into *confused deputies*. This is always the case if a trusted system class uses untrusted input to perform reflective access, e.g., a system class that calls arbitrary method handles originally provided by untrusted application code, or field accesses to fields whose names are provided by untrusted code. Exploits can use such confused deputies to access members that should be inaccessible to them [20].

Type confusion – Java was designed to provide strong type-safety guarantees. The platform combines static and dynamic checks to prevent that an object of one type is illegally cast to an incompatible type. Enforcing type safety is complex, however, and in the past, several defects made the platform susceptible to type-confusion attacks.

In a type-confusion attack, the exploit causes the JVM to incorrectly treat an object of a certain type `A` as an object of another type `B`. Consequently, the attacker can perform actions on this object that are allowed for type `B`, but not for type `A`. If, for example, type `A` declares a private field at a certain offset, and type `B` declares a public field at the same position, type confusion can be used to illegally access the private field—convenient, for instance, if one wishes to null out the field `System.security`.

Buffer overflows – The JVM is implemented in native code and hence potentially susceptible to buffer overflows, which attackers may use to alter portions of memory. The consequence of overwriting memory locations this way is different from case to case, and strongly depends on the type of memory locations that are affected by the vulnerability. If the target buffer that is affected by the vulnerability resides on the stack,[1] attackers will typically attempt to overwrite return addresses, which would allow them to take over the control flow of the application. If instead the vulnerable buffer is allocated on the heap,[2] attackers might attempt to manipulate application-specific data structures, as the stack is out of reach. In the context of Java security, this might be the contents of sensitive variables of system classes which are stored on the heap.

[1] see "CWE-121: Stack-based Buffer Overflow"

Due to the different ways in which different buffer overflow vulnerabilities can impact the security of the Java platform, we differentiate two different cases.

Information-hiding attacks use buffer overflows that affect buffers that are near the locations of sensitive variables of system classes, which the JVM stores on the heap. Such attacks can be used to overwrite private variables and hence represent a breach of information hiding. One example of such an attack on the Java platform is the exploit for CVE-2013-2465. These attacks are within the scope of this paper, and the countermeasures presented in this paper are designed to mitigate them.

Control-flow hijacking attacks use buffer overflows that affect memory locations whose integrity is important to the control flow of the application, such as return addresses on the call stack of the JVM. These attacks can take over the application's control flow and hence achieve arbitrary code execution without corrupting information hiding. Control-flow hijacking attacks are not specific to Java, but instead affect a broad range of native applications. There are a variety of countermeasures that have been implemented in the past, such as data execution prevention [30], or address space layout randomization, to address such attacks. Moreover, there is an active research community that develops novel countermeasures, e.g., control-flow integrity checks [2]. Note that the JVM is equally susceptible to control-flow hijacking attacks as, e.g., PDF viewers, browsers, or other native applications. This means that a solution that mitigates such attacks for Java is likely a general solution to the problem.

Since the focus of this paper is on strengthening information hiding in Java, we consider control-flow hijacking attacks as out of scope. However, the countermeasures that were proposed in academia to address the problem of control-flow hijacking can also be implemented for the JVM, even complementary to the countermeasures we propose in this paper to strengthen information hiding.

4 PROOF-OF-CONCEPT SOLUTION

As explained in the introduction, this paper gives a detailed account of *two* possibilities to strengthen information hiding such as to avert all attacks known to date. The *heavyweight* mitigation strategy is clean and simple to explain, and is what we would suggest for productive use. Yet, implementing this solution requires access to the JVM's source code, and significant engineering effort. Section 6 explains the design of this strategy.

In this section, we instead present a backward-compatible prototype solution that goes without modifications of the JVM. It is composed of a systematic orchestration of several countermeasures specifically designed to strengthen information hiding in Java. These countermeasures effectively block a broad range of attacks on information hiding, while retaining backward compatibility and

[2] see "CWE-122: Heap-based Buffer Overflow"

Table 1: Relationship between countermeasures and attack vectors

	Attack vectors		
Countermeasure	Introspection	Type conf.	Buffer overfl.
Field blackbox	✓	✓	✓
Integrity checker	(✓)*	(✓)*	(✓)*
Method blackbox	✓	✓	-
Classpool layout randomization	-	✓	✓

*partially: ensures field values' integrity, but not confidentiality

high performance. We designed them such that they can be instrumented into even closed-source JREs, which allows us to perform tests on different platforms with different versions.

However, as we explain in detail below, these advantages come at the cost of using secret tokens that our proof-of-concept solution hardwires into the method bodies of system classes. As we explain, these tokens are randomly generated and individual for every host system. It would be possible to generate and update the secret tokens during the installation routine of the JRE, or even before every single application run. Alternatively, it would also be possible to modify the JRE's class loading mechanism in such a way that it automatically generates and integrates new secret tokens into system classes while they are loaded by the JVM. This way, secret tokens would never be persisted on disk, which would further complicate attacks. Importantly, however, this drawback concerns only our experimental lightweight solution. The solution we propose for productive use in Section 6 does not depend on any secret tokens.

4.1 Conceptual overview

Section 3.2 presents the various attack vectors that attackers can use to break information hiding: insecure use of introspection, type confusion, and buffer overflows. In order to address these attacks, we propose the "blackbox": a combination of four countermeasures that can be integrated into the JCL to prevent illegal member access. The following details these countermeasures and their individual purposes in defending information-hiding attacks. Table 1 provides an overview of how the countermeasures relate to the different attack vectors.

(1) *The **field blackbox** restricts sensitive fields to be accessed only from known, trusted code* – The values of sensitive variables of system classes are stored in an isolated storage area to which introspection APIs have only limited access.

(2) *The **method blackbox** restricts sensitive methods to be accessed only from known, trusted code* – Its purpose is to prevent exploits from calling private methods of system classes using introspection APIs or type confusion.

(3) *Some sensitive fields are read (never written!) directly by the JVM. The **integrity checker** assures their integrity, without requiring JVM modifications* – To this end, it adds additional integrity checks to the JCL that serve as security gates, illegal field modification is thus detected before having an effect.

(4) ***Classpool layout randomization** is an additional defense layer against type confusion and buffer overflows* – It randomizes the location of field values in memory by modifying the classpool. This measure alone is not sufficient as countermeasure, as it may be bypassed by advanced search routines. However, it complements the field blackbox and integrity checker by adding an additional layer of security, thus raising the bar for future attacks, and contributing to defense in depth.

4.2 Detailed overview

Field blackbox – As many different attack vectors rely on illegal access to private fields in system classes, protecting sensitive fields is a key feature of the blackbox. We consider all private fields in system classes of the following types as sensitive: `Unsafe`, `SecurityManager`, `AccessControlContext`, `ProtectionDomain`, and `PermissionCollection`, including their array representations. We derived this list of relevant types from a thorough review of the Java security architecture [17], as well as from reviews of exploits in our sample set.

To protect sensitive field values, we replace all regular accesses to them by calls to the blackbox, thus ensuring that they will never be read or written from their original locations. This protects against type confusion and buffer overflows, because sensitive values are no longer stored along with the parent object's representation in memory, but rather in a hard to access location in the blackbox. Conceptually, the blackbox can use various measures to protect its contents, including, but not limited to, memory randomization, on-the-fly encryption, process separation, and even hardware separation. The CHERI ISA [33], for example, is an instruction set that provides dedicated support for hardware-based, fine-grained memory protection. Using the CHERI ISA allows applications to securely restrict how code can access certain memory locations, which could be used by a blackbox implementation to prevent illegitimate access to the values it contains. As we explain in Section 8, Chisnall et al. [8] applied the CHERI ISA to implement a sandboxing mechanism for native code in Java, which highlights the relevance of this instruction set for Java platform security.

Listing 1 shows an example of a method `sysMethod()` that accesses a field `acc` of type `AccessControlContext` in its parent class. Before our modification, `sysMethod()` reads and writes directly the static field (lines 6-7). After integrating the field blackbox into the class library, rather than loading and storing values in `acc`, `sysMethod()` uses the blackbox's public interface to read and write sensitive values instead (lines 9-10). To access field values in the blackbox, `sysMethod()` passes at least two arguments. The first argument, here *2583*, is a secret token required to access *any* security-relevant functionality in the blackbox. Importantly, the runtime automatically protects the secret token much better than the original sensitive field, as it is hardcoded into the method body of the system class, which cannot be inspected or analyzed by untrusted code. The secret tokens cannot be guessed: Any attempt to access a blackbox method without the correct token is considered an attack, and will result in a call to `Blackbox.panic()`. This method is an emergency method that adequately responds to an attack, e.g., by generating logs and reports for manual interventions, or even by terminating the execution of a violating application. Note that the

Listing 1: Effects of integrating the field blackbox into the class library.

```java
// sensitive field
private static AccessControlContext acc;

public void sysMethod(AccessControlContext newAcc) {
  // \\original code:\\
  // acc = newAcc;
  // System.out.println(acc);
  // \\modified code:\\
  Blackbox.set_static_ref(2583,87790L,newAcc);
  System.out.println(Blackbox.get_static_ref(2583,87790L));
}
```

Listing 2: Secure reflective access to sensitive fields will be automatically rerouted to the blackbox to ensure backward compatibility.

```java
1  public Unsafe secReflect() {
2    Field f = Unsafe.class.getDeclaredField("theUnsafe");
3    f.enableBlackbox(2583,383970L);
4    f.setAccessible(true);
5    return (Unsafe)f.get(null); // succeeds
6  }
7
8  public Unsafe insecReflect(String s) {
9    Field f = Unsafe.class.getDeclaredField(s);
10   f.setAccessible(true);
11   return (Unsafe)f.get(null); // fails
12 }
```

solution we propose in Section 6 to strengthen information hiding for productive use does not depend on such secret tokens. Instead, it extends the JVM instruction set by instructions that can only be used by system classes, which, however, requires modifications to the JVM. The second argument, here *87790L*, is a unique identifier associated with the field to be accessed.

Replacing sensitive field accesses in system classes as in Listing 1 causes sensitive fields such as `acc` to become "dead" as they will not any more be read or written by their parent class. Without any further modifications, this would cause legacy reflective accesses to such fields to misbehave, because the reflection API would access the original, now "dead" field, while the variable's parent class would use the field blackbox instead.

Hence one should seek to support reflective accesses to the blackbox. Yet, while doing so, one must be mindful of the problem of reflection-based confused-deputy attacks. Confused-deputy vulnerabilities are caused by system classes that make insecure use of reflection—they call methods of the reflection API in a way that allows attackers to arbitrarily specify the targets of reflective access, thus providing access to private variables of system classes. A secure use of reflection, in contrast, is performed using constant arguments, which cannot be modified dynamically. We found actual examples of system classes that make secure use of reflection to access sensitive fields of other system classes. Preserving this legitimate access is thus required to retain backward compatibility. We did not find any examples of classes in the JCL which make insecure use of reflection to access sensitive fields of other system classes.

We thus propose to extend the reflection API and modify call sites accordingly such that secure, i.e., restricted, uses of reflection will be automatically rerouted to the field blackbox, while insecure, i.e., unrestricted, uses of reflection remain unchanged. To detect relevant call sites, we statically analyze all system classes in the JCL to find calls to `Class.getDeclaredField(fieldName)`. For every such finding, the analysis determines if the call is performed on constant values for `Class` and `fieldName`. As the attacker cannot modify such calls, they imply a restricted, secure use of reflection. Actions performed on an instance of `Field` obtained this way can be safely rerouted to the field blackbox. To achieve this, the analysis instruments any secure call site for `getDeclaredField()` such that the returned instance of `Field` will be permanently associated with the unique identifier for the concerning field. This identifier is required to access the corresponding value in the field blackbox. Our modification of the reflection API causes methods such as `Field.get()` to check for such an association of a unique identifier. If such an association exists, rather than attempting to read the respective value from the original variable, the identifier is used to obtain it from the field blackbox. If the static analysis finds a call site that it does not consider to be secure, or it detects a secure call site that targets a non-sensitive field, the original code will not be modified and the method call will not be rerouted to the blackbox.

Listing 2 illustrates how the analysis deals with different uses of reflection. In this example, `secReflect()` invokes `getDeclaredField()` in line 2 on constant values `Unsafe.class` and `"theUnsafe"`, which the analysis considers to be a secure use of reflection. Because of that, and the fact that `Unsafe.theUnsafe` is a sensitive variable, the analysis will reroute the field access to the blackbox.

The second method in this example, `insecReflect()`, is treated differently. In contrast to `secReflect()`, it contains a call to `getDeclaredField()` in line 9 that passes a variable as field name, rather than a constant. The analysis considers this insecure and does not modify the call site to reroute `Field.get()` in line 11 to prevent potential confused deputy attacks.

Method blackbox – Certain private methods in system classes can be used to bypass security checks. The purpose of the method blackbox is to prevent attackers from using type confusion or reflection to call these sensitive methods.

As illustrated in Listing 3, our approach is simple, and yet effective against common attacks. In this example, `getPrivateFields()` is a sensitive private method that must be protected from illegal invocation. To achieve this, the method blackbox adds an additional argument `token` to `getPrivateFields()` that will be checked at the beginning of the method, see lines 2-3 in Listing 3. The method will continue its normal operations only if `token` contains a specific value, here 9377, otherwise, `Blackbox.panic()` will be called, which serves as an emergency method that adequately responds to

Listing 3: The method blackbox protects private methods by adding a check for a secret token.

```
 1  private Field[] getPrivateFields(int token,
        Class c) {
 2    if(token != 9377) {
 3      Blackbox.panic();
 4    } // ... original code
 5  }
 6
 7  public boolean hasPrivateFields(Class c) {
 8    if(getPrivateFields(9377, c).length > 0)
 9      // ... original code
10  }
```

attempted attacks. As can be seen in line 8, all callers in the parent class will be updated such that they pass the required secret token as an argument, backward compatibility is thus retained. Exploits that obtain access to methods protected this way, e.g., by using a confused deputy, cannot call them without the correct token. Importantly, the solution we propose for productive use in Section 6 does not use any secret tokens. Instead, it extends the JVM instruction set by dedicated instructions for sensitive method calls that only system classes can use.

Integrity checker – The field blackbox is the core feature of our proposed solution, but there are some sensitive fields in the JCL to which it cannot be applied. Those fields are accessed not only by their parent class, but also directly by native code in the JVM. Replacing field accesses only in their parent classes while leaving accesses in the JVM unmodified would result in inconsistent behavior. However, modifying the JVM accordingly would require access to its source code, thus violating the design goals of our proof of concept (see Section 4).

A review of the relevant fields reveals that in all cases the confidentiality of their values is not an issue, but only their integrity is important to the security of the platform. The array `AccessControlContext.context` is an example of such a field. It contains the set of protection domains that will be considered when a permission check is to be executed in the respective context. Its content is not secret and could possibly even be obtained through other means, so it is not required to protect the confidentiality of `context`. Its integrity, however, is of high importance, because illegal modification would compromise access control.

Further, the JVM writes to these fields only during object initialization, if at all. In all other cases, the JVM only performs read accesses, e.g., reading `AccessControlContext.context` when a permission check is to be executed.

To protect also these special fields, we developed the *integrity checker*, an alternative countermeasure that is used to guard sensitive fields that cannot be moved to the blackbox. The basic concept of the integrity checker is to track throughout program execution all legal field modifications of protected fields such that illegal modifications can be detected before they impact the security of the platform. To achieve this, we keep a checksum in the field blackbox for every object that contains at least one field whose integrity is to be protected. Every legal field modification in a parent class will cause the corresponding checksum in the field blackbox to be updated. To prevent attackers from simply updating themselves the checksum after an illegal field modification, updating this value requires a secret token known only to the parent class. Similarly to the field blackbox, this token is stored only within the class' code, and the solution we propose in Section 6 for productive use works without secret tokens.

We added integrity checks to the JCL that serve as security gates—whenever they are reached during program execution, they will evaluate for relevant objects whether the current field values of protected fields still correspond to the checksum in the field blackbox. If this is not the case, at least one of the fields must have been modified illegally, and the blackbox will call its emergency method `Blackbox.panic()`, which adequately responds to attempted attacks.

Classpool layout randomization – Every class file contains a section known as the *classpool*, which lists all class and instance fields of the type. The location of fields in memory is dependent on their position in the classpool. Past exploits expect certain sensitive fields to be at certain memory locations, because their location in the classpool is known.

To break assumptions of known exploits and increase the effort required to implement new low-level exploits, we apply classpool layout randomization. This countermeasure shuffles the order of existing fields in the classpool and adds a random number of new dummy fields, which results in randomized locations of field values in memory. All attacks that aim for specific fields in memory will thus have to integrate a search routine to find the correct locations. We currently apply this countermeasure only to `AccessControlContext`, but it can be applied to other system classes as well.

4.3 Implementation

To be able to validate our prototype implementation of the proposed countermeasures on different JREs, we designed it as a transformation engine. It takes as input the class library of a given JRE (e.g., the rt.jar file of the Oracle JRE), then modifies all required classes to integrate our proposed countermeasures, and finally outputs the set of files that have changed. Then any original, unmodified JVM can be instructed via command line arguments to use the modified system classes that were generated by the transformation engine.

Our implementation uses the Javassist [7] library to locate fields and methods, and to implement all modifications to class files that are necessary to integrate our proposed countermeasures into the JCL. We use ASM [1] to identify system classes that implement statically-confined reflective accesses to fields protected by the blackbox, so that we can redirect these calls accordingly.

The transformation engine is implemented such that it automatically searches for fields and methods that meet our specified criteria, which significantly reduces the effort required to adjust the engine to previously unseen JRE implementations. This high portability allowed us to apply it to Oracle Java 7 and 8 on Windows, as well as to IBM Java 7 on Linux.

The prototype features two modes of operation. In one of them, all sensitive values are stored in native code, all accesses have to be made through JNI. Since this is implemented in native code, the technical measures that could be used to protect the blackbox and its contents are unlimited. In the other mode, sensitive values are stored in a dedicated Java class, which is only used within methods of system classes that legally access the blackbox. This mode of operation is highly portable as it is implemented in Java, but its capabilities to use certain protection mechanisms are also restricted.

4.4 Limitations

One major concern of the lightweight solution is the use of secret tokens that are hardwired into the method bodies of system classes. The purpose of these tokens is to prevent untrusted code from accessing methods of the blackbox. The use of secret tokens in our proof of concept is the result of a deliberate design choice: it is a tradeoff that allows us to integrate our lightweight solution into closed-source Java runtimes without having to modify native components, at the cost of reduced attack resistance. The heavyweight solution we propose for productive use in Section 6 requires no secret tokens, but, as we will explain, its implementation requires modifications to the JVM.

Other concerns, besides the use of secret tokens, relate to the storage area of the blackbox, that is supposed to securely contain the contents of security-critical variables. Similar to the secret tokens, also these values might potentially be leaked to attackers due to memory safety problems. One mitigation strategy is to leverage software-based protection mechanisms, such as process isolation, or ASLR. Using Rust [24] to implement the native blackbox components would provide strong guarantees with respect to memory safety, thus reducing the risk of introducing security-critical vulnerabilities. Additional mitigation strategies involve the use of dedicated hardware support. TPMs can be used to store sensitive values, which would significantly strengthen the implementation of the blackbox. Alternatively, as we explained previously in Section 4, the CHERI ISA [33], or similar solutions can be used to prevent untrusted code from accessing sensitive memory locations.

5 EVALUATION

We next explain how we used our "lightweight" prototype to evaluate the following research questions:

RQ1: How effective is the blackbox in blocking existing attack vectors?
RQ2: Is the blackbox backward compatible with legacy applications?
RQ3: What is the performance impact of the blackbox on applications?

5.1 RQ1: Effectiveness

We evaluated the effectiveness of the blackbox using a sample set of 61 unique, well-studied exploits [21], which, to the best of our knowledge, is the most comprehensive sample set of Java exploits available. We added an additional exploit that we forked from one of the 61 exploits to demonstrate that violating the integrity of a field that is accessed directly by the JVM is sufficient to compromise the Oracle JRE.

As a first step, we removed all exploits that do not depend on a breach of information hiding, as they are out of scope. The result is a set of 38 relevant exploits, which is a larger share than we originally expected. We then evaluated how many of these 38 information-hiding attacks are blocked by our proof-of-concept solution. We found that our solution effectively blocks 32 exploits (84%). Table 2 provides details on the effectiveness of each of the four countermeasures. Considering that under normal conditions Java's regular protection of private class members is also in effect, all 32 exploits are now mitigated by at least two complementary countermeasures, five of them even by three.

Finally, we reviewed the remaining six information-hiding attacks not blocked by our solution, and found that they implement the same attack vector, but using different vulnerabilities as a foundation. All these attacks use defects that provide illegitimate access

Table 2: Overview of the exploits that are blocked by at least one of the countermeasures that we propose.

		Countermeasures			
	Exploit	Fields	Integrity	Methods	Layout
Oracle JRE on Windows	IntegrityTestExploit	✓	✓	-	-
	NO-CVE-27	✓	-	-	✓
	MULTI-CVE-2012-5075-2012-4681	✓	-	-	-
	MULTI-CVE-2012-4681-2012-5074	✓	-	-	-
	MULTI-CVE-2012-1682-2012-1726	✓	-	-	-
	MULTI-CVE-2012-0547-2012-1726	✓	-	-	-
	CVE-2013-2465	✓	-	-	-
	CVE-2013-1475	✓	-	-	✓
	CVE-2012-1726	✓	-	✓	-
	CVE-2012-5076b	✓	-	-	-
	CVE-2012-5076c	✓	-	-	-
	CVE-2012-5076d	✓	-	-	-
	CVE-2012-5076e	✓	-	-	-
	CVE-2012-5076f	✓	-	-	-
	CVE-2012-5076g	✓	-	-	-
	CVE-2013-2423	✓	-	-	-
	CVE-2012-4681	✓	-	-	-
	CVE-2012-5076a1	✓	-	-	-
	NO-CVE-3	✓	-	-	-
IBM JDK on Ubuntu	NO-CVE-8-ibm	✓	-	-	-
	NO-CVE-9-ibm	✓	-	-	-
	NO-CVE-10-ibm	✓	-	-	-
	NO-CVE-11-ibm	✓	-	-	-
	NO-CVE-12-ibm	✓	-	-	-
	NO-CVE-13-ibm	✓	-	-	-
	NO-CVE-14-ibm	✓	-	-	-
	NO-CVE-18-ibm	✓	-	-	-
	NO-CVE-19-ibm	✓	-	-	-
	NO-CVE-20-ibm	✓	-	-	-
	NO-CVE-24-ibm	✓	-	-	✓
	NO-CVE-26-ibm	-	✓	-	-
	NO-CVE-30-ibm	-	✓	-	-

to `ClassLoader.defineClass()`, a protected method that attackers can abuse to define high-privileged, custom classes. The method blackbox is not applied to this method to remain backward compatible: when there is no security manager, there may be user-provided subclasses of `ClassLoader` that call `defineClass()` for legitimate purposes. While our current prototype does not block these attacks, there are two possible solutions to address this.

One solution is to apply the method blackbox also to `defineClass()` in cases when a security manager is set. While our current implementation does not consider the status of the security manager, such support can be added for this purpose. Another solution is to apply the "privilege escalation rule" proposed by Coker et al. [11], which prevents classes from loading other classes with higher privileges. It was designed to mitigate attacks involving `defineClass()` or similar functionality. Although this countermeasure does not address information-hiding attacks in general, it is an ideal complement to our solution.

5.2 RQ2: Backward compatibility

To evaluate the backward compatibility of the blackbox, we executed a small set of simple test applications, as well as all benchmarks in the DaCapo benchmark suite [6] version 9.12-bach on our modified versions of the Oracle JRE 7 and IBM JDK 7. The DaCapo suite consists of 14 different benchmarks, which represent complex real-world applications of various application domains, running non-trivial workloads. All applications completed as expected and we did not discover any misbehavior that resulted from our modifications. The countermeasures we propose are highly backward compatible. Integrating the blackbox into the JCL does not modify any public interfaces, the officially supported functionality of the platform is retained, and applications run unaffectedly.

The prototype is implemented as a transformation engine that instruments its changes into an original JRE installation. It can be applied to a broader spectrum of different JREs, in different versions, on different platforms, although JVM-specific adaptations may be required, e.g., concerning the fields that are to be protected by the integrity checker.

5.3 RQ3: Performance

We used the DaCapo benchmark suite for a large-scale performance evaluation, because it was specifically designed for this purpose [6].

We executed all benchmarks under several configurations. First, we run all benchmarks using an unmodified Oracle JRE 7 installation, which serves as a baseline for comparisons. Then, we run all benchmarks on the modified Oracle JRE 7 that contains the blackbox mechanisms. This is performed twice per benchmark: once in the "Java blackbox" mode of operation, which stores sensitive values in a dedicated Java class, and once in the "native blackbox" mode of operation, which stores sensitive values in a custom native component. Finally, all tests are executed two times, once without a `SecurityManager`, and once with the default `SecurityManager` with a security policy that grants all permissions to all code. In each configuration, we conducted 500 timed application runs per benchmark. We excluded the first 100 runs, which served only as a warmup, and reported the average of the remaining runs as a result. Measurements were performed on Windows 10, Intel Core i5-3350P 3.10GHz CPU, with 12GB RAM.

Table 3: Results of our performance measurements. Absolute overheads provided in milliseconds.

	Java blackbox				Native blackbox			
	w/o SM		w/ SM		w/o SM		w/ SM	
Benchmark	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.
avrora	-23	-1%	-4	0%	-4	0%	-16	-1%
batik	-2	0%	-*	-*	1	0%	-*	-*
eclipse	437	3%	1925	10%	138	1%	2412	13%
fop	-5	-3%	-5	-2%	2	1%	-1	-1%
h2	-90	-2%	-79	-2%	8	0%	143	4%
jython	19	1%	31	2%	24	2%	47	3%
luindex	3	0%	2	0%	-25	-3%	35	3%
lusearch	-4	0%	-22	-2%	-21	-2%	5	1%
pmd	32	2%	58	3%	75	4%	113	6%
sunflow	-6	0%	-41	-2%	-9	0%	-21	-1%
tomcat	-113	-1%	295	2%	-225	-2%	1019	5%
tradebeans	-15	0%	21	1%	8	0%	72	2%
tradesoap	445	2%	-119	0%	32	0%	-337	-1%
xalan	11	1%	2	0%	-5	-1%	-9	-1%
Geom. mean		0%		1%		0%		2%

*`batik` cannot be executed with a security manager.

Table 3 shows the results of our performance tests. As can be seen, in the "Java blackbox" mode of operation, all overheads lie in the range of -3% and 10%. Except for two outliers, overheads induced by the blackbox are between -2% and 3%. Only seven benchmarks show a slowdown at all. Negative overheads imply that our modifications increased execution speed, however, this is rather unlikely. Our measurements are subject to small inaccuracies, which are not caused by deficiencies in our experimental setup, but rather inherent to all Java benchmarks executed on a real hardware/software stack. Gil et al. [16] studied this effect in depth and found that even executing identical code can lead to slightly different runtime measurements. Gu et al. [19] identified various factors that can lead to variances in execution speed of Java applications.

The performance measurement results for the "native blackbox" mode of operation are similar, but we observe slightly higher overheads for some benchmarks. Overall, the overheads lie between -3% and 13%. Excluding two outliers, overheads range from -2% to 6%. We expected these higher overheads, because the native blackbox involves a higher number of JNI calls, which are costly compared to regular Java calls.

The highest overheads in both modes of operation can be seen for Eclipse. The reason is that this benchmark is itself a collection of performance tests for the Eclipse Java Development Tools, which frequently exercise a specific call sequence that became slower through our modifications. Even though Eclipse is a real-world application, the workload it performs in the DaCapo benchmark suite does not at all resemble typical user behavior.

In summary, average overheads (geometric means) lie between 0% and 2%.

6 REDESIGNING JAVA FOR PRODUCTIVE USE

The following presents ideas on how to strengthen information hiding for productive use, overcoming weaknesses of our proof-of-concept approach.

The first step in improving the security design of the platform would be to consolidate the core of policy enforcement in a central, isolated component, which we call the *Java security monitor*. This includes all data structures that determine the security status of the system (e.g., whether security mechanisms are enabled or not), data structures involved in policy enforcement (security policies, etc.), and all algorithms that operate on these data structures.

For protecting sensitive members in system classes that do not belong to the core of policy enforcement (like `Statement.acc`), we propose to extend the Java Language Specification [18] by an additional modifier `critical`. Members modified this way should be protected in the same way as the blackbox protects fields: reflective access should, if at all, only be allowed when performed in a statically-confined manner, and the member's location in memory should be randomized to prevent low-level attacks. Further, we propose to extend the JVM instruction set by a new set of instructions whose use is reserved for system classes only. These instructions, on the one hand, comprise dedicated commands that system classes must use to call private methods. On the other hand, the new instruction set comprises dedicated commands for `critical` field access. We provide a specification of these instructions in Appendix A. Since application classes cannot use these instructions, they are systematically prevented from accessing sensitive members directly, or by means of reflection. This design prevents illegitimate access to sensitive members without the use of secret tokens, which reduces the attack surface compared to the proof-of-concept implementation presented in Section 4.

We suggest the following strategy to implement our proposed changes:

(1) Consolidate the core of policy enforcement in a natively implemented security monitor, which includes all functionality required to carry out permission checks. The security monitor must further comprise a dedicated, secure data store for `critical` field values, like `Statement.acc`. The implementation should be based on a multi-process architecture, such that operating system-based process isolation can be used to separate the security monitor from application code.

(2) Extend the JVM's bytecode interpreter and just-in-time compiler by the new set of instructions that we propose for sensitive member access.

(3) Modify the platform's bytecode verifier to provide support for the new set of instructions. The verifier must ensure that these instructions are only used by system classes, by which we avoid the use of any secret tokens.

(4) Modify the JCL's source code to add the new modifier `critical` to all sensitive fields in system classes that do not belong to the core of policy enforcement, like `Statement.acc`. It is possible to reuse our proof-of-concept implementation to automatically identify all relevant fields.

(5) Modify the Java compiler as follows:

(a) System classes that access a `critical` field must use the extended instruction set for this purpose to make use of the security monitor's data store.

(b) System classes must use the extended instruction set to call private methods.

(c) Extend the compiler by a static analysis that detects reflective access from a system class to `critical` members of other system classes, using a similar approach like our proof-of-concept implementation. Statically-confined, and hence secure accesses to private members must be treated like regular, legal `critical` member accesses and must apply the new instruction set to allow the access.

(6) Extend the runtime to enforce the "privilege escalation rule" proposed by Coker et al. [11], which prevents classes from loading other classes with higher privileges. This specifically guards against attacks that abuse `ClassLoader.defineClass()`, or similar functionality.

7 SECURE SOFTWARE DESIGN

The case of Java raises general questions concerning the secure design of complex systems. Despite significant efforts, many large-scale systems were repeatedly exposed to single vulnerabilities which undermined their entire security architectures. Such fragility must be inherent to the implemented architectures, and continuous efforts in patching individual bugs does not solve the underlying problems in system design.

We can measure the brittleness of a target software system in two dimensions:

Compile-time metric – *What is the minimum number of lines in the target system's source code that need to be modified to compromise the whole system?* In the case of Java, a single line of code is sufficient. Inserting `return null;` in the beginning of `System.getSecurityManager()` is just one example, and there are many alternatives.

Runtime metric – *What is the minimum number of bits in memory that need to be modified to compromise the target system?* For Java, the number of bits is lower or equal to the size of a single object reference in memory. Flipping the bits that represent `System.security` is just one example.

Low numbers for any of the two metrics represent high brittleness, however, it is not clear to what extent these metrics correlate. Besides the obvious examples that we gave to illustrate Java's brittleness, there is an unknown number of non-obvious modifications that would equally compromise policy enforcement. In fact, replacing a single occurrence of `==` by `=` might break a large number of highly security-critical software systems. High brittleness not only implies that a software system is susceptible to security bugs introduced unintentionally, it also implies a high potential for hiding hard-to-find backdoors. Java was designed as a self-protecting system, i.e., the security mechanisms that are supposed to protect the system are part of the system itself. This inherently leads to circular dependencies, which presumably contributes to brittleness. An alternative to a self-protecting system is a system that is compartmentalized such that access control decisions are carried out by a dedicated component, the *reference monitor*. This fundamental concept has its origins in the design of secure military systems, and it aims for reliable and hard-to-bypass policy enforcement.

Early work in this area [4] defines a set of principles, according which the reference monitor has to be tamper proof, be involved in all security-related decisions, and simple enough to be verifiable. The similar concept of a *security kernel* directly associates tamper resistance with isolation [28]. The redesign of Java's security architecture that we proposed is fully in line with these principles. Previous work referred to the `SecurityManager` as an implementation of a reference monitor [15], however, we argue that it does not sufficiently comply to any of the above principles.

The fragility of complex software is, to large extents, caused by shortcomings in its security design. This detailed study of information hiding in Java can be seen as a case study that supports this claim. Further research is needed to provide a better understanding of how to detect and prevent security design flaws.

8 RELATED WORK

Various different approaches to strengthen the Java platform have been proposed in the past. However, as we explain in the following, none of them adequately protect against common attacks on information hiding.

There is a countermeasure that specifically addresses the dangers of insecure use of reflection: so-called *filter maps*. System classes can use public interfaces of the reflection API to add individual class members to filter maps, which hides them from reflective access, similar to a blacklist. `System.security` is an example of a field that is added to the field filter map by default—calling `System.getDeclaredFields()` will thus return all declared fields except for `security`. This countermeasure is generally ineffective against low-level attacks involving, e.g., type confusion, as sensitive field values remain in the same memory locations. Filter maps could, in theory, be effective against attacks involving confused deputies that insecurely use reflection, but in practice, many sensitive fields are not added to filter maps.

Besides, there have been various proposals in academia to revise Java's original approach to stack inspection [3, 25, 32]. While these approaches overcome certain shortcomings of Java's stack inspection routine, they were not designed to strengthen information hiding. Especially low-level attacks involving type confusion and buffer overflows are not in scope of these mechanisms, and they hence do not provide sufficient protection against these attacks.

Various techniques have been proposed to detect and prevent buffer overflows in native code [12, 13, 23, 27], including approaches to enforce control-flow integrity [2, 34]. Although this field of research advanced in the past years, buffer overflows are still considered a challenge today, as many approaches suffer from practical problems, like high performance overheads, or low precision. Moreover, none of these techniques address illegal field access in Java through type confusion, or by means of a confused deputy.

Classpool layout randomization is a countermeasure that achieves randomized memory locations of sensitive fields of Java classes. Different variants of address obfuscation and randomization have been proposed and implemented in the past [5, 22, 29], however, to the best of our knowledge, we are the first ones to implement this specifically for Java class files.

Woodruff et al. [33] presented the CHERI Instruction-Set Architecture (ISA), which implements fine-grained, capability-based memory protection at the hardware layer. Our prototype implementation of the blackbox uses software-based mechanisms and secret tokens to prevent illegal access to the contents of the blackbox. The CHERI ISA could be used to separate the contents of the blackbox and restrict access to only legitimate callers on the hardware level, thus further strengthening the implementation of our proposed concept. Chisnall et al. [8] used the CHERI ISA to implement a sandboxing mechanism for native code in Java, which ensures that also this code adheres to Java's security and memory model, thus preventing access to security-critical memory locations.

Toledo et al. [31] investigated a different approach to redesign access control in Java. They found that security-related code is scattered all over the codebase, and use aspect-oriented programming to modularize access control. Their solution increases maintainability and allows for finer-grained policies, but it does not address common attacks on information hiding.

Finally, it is important to note that the Java module system [26] was introduced with Java 9. This new scheme allows for stronger encapsulation of sensitive system classes, resulting in fewer system classes being available to attacker-controlled code, which potentially reduces the attack surface. However, the module system is not specifically designed to mitigate the low-level attacks we discussed in this paper, see Section 3. For example, illegal field modification through a type confusion vulnerability or buffer overflow remains an important issue that is not solved by the module system. Further, as shown by Dann et al. [14], the module system does not comprehensively prevent sensitive objects from escaping their declaring modules, even in cases where they are declared to be internal, which can result in vulnerabilities similar to the ones we discussed here. From this, we conclude that the countermeasures we propose remain highly relevant. From a conceptual point of view, the solutions we present in this paper are compatible with the module system, and implementing the solution we propose in Section 6 together with the module system may in fact provide the strongest defense.

In summary, we conclude that no countermeasure that has been proposed previously reliably and comprehensively protects against common attacks on information hiding in Java.

9 CONCLUSION

In this paper, we studied the role of information hiding in Java's security model and found that a large portion of common attack vectors depend on illegal access to private members of system classes. We presented a proof-of-concept solution that significantly strengthens information hiding in the Java platform by combining a set of different countermeasures that can be integrated into the JCL. An evaluation shows that these countermeasures block 84% of all information-hiding attacks in a large-scale sample set, and we explain how even the remaining attacks can be blocked as well. We further show that our solution retains backward compatibility and high performance—a set of real-world applications runs with an average overhead below 2%. Moreover, we suggest a fundamental redesign of the Java platform for productive use that has the potential to outperform our proof of concept in terms of robustness and speed. We finally discuss general thoughts on the design of secure software, which are based on the insights we gained during our study.

REFERENCES

[1] [n.d.]. https://asm.ow2.io/.

[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security.* ACM, 340–353.

[3] Martin Abadi and Cédric Fournet. 2003. Access Control Based on Execution History.. In *NDSS*, Vol. 3. 107–121.

[4] James P Anderson. 1972. *Computer Security Technology Planning Study. Volume 2.* Technical Report. DTIC Document.

[5] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. 2003. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits.. In *USENIX Security Symposium*, Vol. 12. 291–301.

[6] 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. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[7] Shigeru Chiba. 1998. Javassist - a reflection-based programming wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java (Vol. 174).*

[8] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A Theodore Markettos, J Edward Maste, Robert Norton, Stacey Son, et al. 2017. CHERI JNI: Sinking the Java security model into the C. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 569–583.

[9] Cisco. 2013. 2013 Cisco Annual Security Report. http://www.cisco.com/web/offer/gist_ty2_asset/Cisco_2013_ASR.pdf.

[10] Cisco. 2014. 2014 Cisco Annual Security Report. http://www.cisco.com/web/offers/lp/2014-annual-security-report/index.html.

[11] Zack Coker, Michael Maass, Tianyuan Ding, Claire Le Goues, and Joshua Sunshine. 2015. Evaluating the flexibility of the Java sandbox. In *Proceedings of the 31st Annual Computer Security Applications Conference.* ACM, 1–10.

[12] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.. In *Usenix Security*, Vol. 98. 63–78.

[13] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2008. Real-World Buffer Overflow Protection for Userspace and Kernelspace.. In *USENIX Security Symposium.* 395–410.

[14] Andreas Dann, Ben Hermann, and Eric Bodden. 2019. ModGuard: Identifying Integrity &Confidentiality Violations in Java Modules. *IEEE Transactions on Software Engineering* (2019).

[15] Drew Dean, Edward W Felten, and Dan S Wallach. 1996. Java security: From HotJava to Netscape and beyond. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on.* IEEE, 190–200.

[16] Joseph Yossi Gil, Keren Lenz, and Yuval Shimron. 2011. 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, 297–308.

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

[18] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2014. The Java Language Specification, Java SE 8 Edition (Java Series).

[19] Dayong Gu, Clark Verbrugge, and Etienne M Gagnon. 2006. Relative factors in performance analysis of Java virtual machines. In *Proceedings of the 2nd international conference on Virtual execution environments.* ACM, 111–121.

[20] Philipp Holzinger, Ben Hermann, Johannes Lerch, Eric Bodden, and Mira Mezini. 2017. Hardening Java's Access Control by Abolishing Implicit Privilege Elevation. In *2017 IEEE Symposium on Security and Privacy (Oakland S&P). IEEE, IEEE Press. To appear.*

[21] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. 2016. 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, 779–790.

[22] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual.* IEEE, 339–348.

[23] David Larochelle, David Evans, et al. 2001. Statically Detecting Likely Buffer Overflow Vulnerabilities.. In *USENIX Security Symposium*, Vol. 32. Washington DC.

[24] Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

[25] Marco Pistoia, Anindya Banerjee, and David A Naumann. 2007. Beyond stack inspection: A unified access-control and information-flow security model. In *Security and Privacy, 2007. SP'07. IEEE Symposium on.* IEEE, 149–163.

[26] Mark Reinhold. 2017. The Java Platform Module System (JSR 376). http://cr.openjdk.java.net/~mr/jigsaw/spec/.

[27] Olatunji Ruwase and Monica S Lam. 2004. A Practical Dynamic Buffer Overflow Detector.. In *NDSS*, Vol. 2004. 159–169.

[28] Roger R Schell, Peter J Downey, and Gerald J Popek. 1973. *Preliminary Notes on the Design of Secure Military Computer Systems.* Technical Report. DTIC Document.

[29] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security.* ACM, 298–307.

[30] Nenad Stojanovski, Marjan Gusev, Danilo Gligoroski, and Svein J Knapskog. 2007. Bypassing data execution prevention on microsoftwindows xp sp2. In *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on.* IEEE, 1222–1226.

[31] Rodolfo Toledo, Angel Nunez, Eric Tanter, and Jacques Noyé. 2012. Aspectizing Java access control. *IEEE Transactions on Software Engineering* 38, 1 (2012), 101–117.

[32] Dan S Wallach, Andrew W Appel, and Edward W Felten. 2000. SAFKASI: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9, 4 (2000), 341–378.

[33] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on.* IEEE, 457–468.

[34] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on.* IEEE, 559–573.

A EXTENSION TO THE JVM INSTRUCTION SET

Instruction:
getfield_critical

Operation:
Fetch critical field from object.

Format:

```
getfield_critical
indexbyte1
indexbyte2
```

Operand stack:

```
..., objectref →
..., value
```

See also:
getfield

Instruction:
getstatic_critical

Operation:
Get critical static field from class.

Format:

getstatic_critical indexbyte1 indexbyte2

Operand stack:

..., → ..., value

See also:
getstatic

Instruction:
putfield_critical

Operation:
Set critical field in object.

Format:

putfield_critical indexbyte1 indexbyte2

Operand stack:

..., objectref, value → ...

See also:
putfield

Instruction:
putstatic_critical

Operation:
Set critical static field in class.

Format:

putstatic_critical indexbyte1 indexbyte2

Operand stack:

..., value → ...

See also:
putstatic

Instruction:
invokespecial_critical

Operation:
Invoke private instance method.

Format:

invokespecial_critical indexbyte1 indexbyte2

Operand stack:

..., objectref, [arg1, [arg2 ...]] → ...

See also:
invokespecial

Instruction:
invokestatic_critical

Operation:
Invoke private static method.

Format:

invokestatic_critical indexbyte1 indexbyte2

Operand stack:

..., [arg1, [arg2 ...]] → ...

See also:
invokestatic