symmetry

MDPI

Article

RIM4J: An Architecture for Language-Supported Runtime Measurement against Malicious Bytecode in Cloud Computing

Haihe Ba [1,†], Huaizhe Zhou [1,†], Huidong Qiao [1,2], Zhiying Wang [1,*] and Jiangchun Ren [1]

[1] College of Computer, National University of Defense Technology, Changsha 410073, China; haiheba@nudt.edu.cn (H.B.); huaizhezhou@nudt.edu.cn (H.Z.); qiaohuidong13@nudt.edu.cn (H.Q.); jcren@nudt.edu.cn (J.R.)

[2] College of Computer and Communication, Hunan Institute of Engineering, Xiangtan 411100, China

* Correspondence: zywang@nudt.edu.cn

† These authors contributed equally to this paper.

Received: 13 June 2018; Accepted: 29 June 2018; Published: 2 July 2018

Abstract: While cloud customers can benefit from migrating applications to the cloud, they are concerned about the security of the hosted applications. This is complicated by the customers not knowing whether their cloud applications are working as expected. Although memory-safety Java Virtual Machine (JVM) can alleviate their anxiety due to the control flow integrity, their applications are prone to a violation of bytecode integrity. The analysis of some Java exploits indicates that the violation results primarily from the given excess sandbox permission, loading flaws in Java class libraries and third-party middlewares and the abuse of *sun.misc.Unsafe*API. To such an end, we design an architecture, called RIM4J, to enforce a runtime integrity measurement of Java bytecode within a cloud system, with the ability to attest this to a cloud customer in an unforgeable manner. Our RIM4J architecture is portable, such that it can be quickly deployed and adopted for real-world purposes, without requiring modifications to the underlying systems and access to application source code. Moreover, our RIM4J architecture is the first to measure dynamically-generated bytecode. We apply our runtime measurement architecture to a messaging server application where we show how RIM4J can detect undesirable behaviors, such as uploading arbitrary files and remote code execution. This paper also reports the experimental evaluation of a RIM4J prototype using both a macro- and a micro-benchmark; the experimental results indicate that RIM4J is a practical solution for real-world applications.

Keywords: Java bytecode; runtime measurement; cloud security; trusted computing

1. Introduction

Cloud computing gives customers a great deal of benefit in comparison with traditional computing. For example, migrating applications to the cloud can remove the burden of repair and maintenance and furnish united availability by way of replication and redundancy [1]. Due to the efficiency and flexibility of the cloud, it has become prevalent to embrace the cloud in both private and enterprise domains. However, security is now the major obstacle for further use of cloud computing [2–4].

This is complicated by the customer not knowing whether his/her cloud applications are working as expected. One of the most dangerous attacks is memory corruption bugs. Malicious attackers exploit these bugs to alter the behavior of the application or even hijacking the control flow. The most obvious solution to avoid these kinds of bugs would be to write applications in type-safe languages [5]. As such, it is always a better option for customers to leverage the safety property of Java platforms to secure their applications in the cloud.

The vendors of Java platforms fulfill the memory-safety property by containing the safe execution of untrusted bytecodes in a so-called sandbox and isolating them from one another. However, some research already has revealed that the sandbox is no longer unbreakable and may be bypassable in recent years [6–9]. An adversary can execute malicious bytecode inside the same protection domain as trusted code or tamper with benign bytecode and even replace it deliberately on the fly.

Through the analysis of related work [6–9] and a set of Java exploits [10–24], we found that there are three factors to account for such a violation of byte integrity. First, the design and engineering of the Java security sandbox is too complex and affords developers more flexibility than they need or use in practice [6]. The unnecessary complexity and flexibility enable attackers to create a self-defined class loader to load malicious classes into any protection domain, such as the exploit of CVE-2012-0507 [10]. Second, a variety of loading flaws [7,8] in the Java class libraries and third-party middlewares also pose a risk to the security of the Java platform. They allow attackers to execute arbitrary bytecode by sandbox-bypassing exploits, such as vulnerabilities in the Java class libraries [11,12] and third-party middlewares [13–23]. Last but not least, many third-party frameworks have extensively utilized the powerful, but dangerous API *sun.misc.Unsafe* to write high-performance code in Java [9] without the necessary security checks. For example, in the exploit of CVE-2012-5076 [24], an attack bypasses arbitrary permission checks to execute malicious bytecode in the memory.

The load-time measurements [25–28] can rely on system call interception to measure bytecode integrity, but this alone does not accurately reflect runtime behaviors. Even worse, the operating system (OS) views Java bytecode as data and manages the bytecode through writable heap areas. As such, we cannot achieve runtime integrity by leveraging the natural support of all modern processors to set all memory pages containing read-only bytecode. Although various runtime measurement approaches [29–31] provide the integrity of running applications through virtual machine introspection, these approaches all introduce a new challenge: portability (i.e., could be used on a variety of Java Virtual Machines (JVMs) and platforms). Such approaches have limited applicability due to the need for recompilation of the operating system, the need for a specialized language-level virtual machine or the need to access application source code. Moreover, these approaches, as mentioned above, cannot work well in the measurement of dynamically-generated bytecode as the type of bytecode is directly created in the memory by the JVM. As such, these existing approaches cannot capture the measured target by using the system call interception. Furthermore, they cannot leverage virtual machine introspection due to the semantic gap in how to accurately extract the semantic meaning about the bytecode from the outside view of the JVM.

Our goal in this paper is two-fold: (1) to build a portable runtime measurement system to be applied seamlessly on other platforms; (2) to enforce an integrity measurement of dynamically-generated bytecode to reduce the likelihood of false negatives. Our key idea is to take advantage of the virtual machine features (attachment and instrumentation) supported by commonly-used JVMs from vendors such as Oracle and the OpenJDKproject, ensuring a runtime measurement without requiring any modification to the underlying client systems (e.g., hypervisors, operating systems, language-level virtual machines).

To such an end, we propose a portable architecture toward the integrity of Java applications within a cloud environment, called RIM4J, to enforce a runtime measurement, with the ability to attest this to a cloud customer in an unforgeable manner. RIM4J performs fine-grained measurement at the level of Java classes (a class containing Java bytecode is the basic unit for the implementation of application behaviors). Instead of measuring the entire content of a class, RIM4J measures only the critical piece of a class (bytecode) executed on the JVM. RIM4J also applies various measurement modes (one-time or periodic) in light of an attestation request from a cloud customer. These modes are used to seek a balance between performance overhead and security assurance. In such a way, RIM4J builds an integrity measurement system for Java bytecode under various modes, further enabling an attestation with acceptable overhead. Our contributions of this paper are summarized as follows:

- A novel scheme to provide a runtime measurement for Java applications within a cloud environment by taking advantage of commonly-used JVMs-supported attachment and instrumentation features.
- A distinction between the dynamically-generated classes and normal loaded classes to reduce the likelihood of false negatives by leveraging virtual machine-specific language features. To the best of our knowledge, RIM4J is the first measurement approach for the integrity of the dynamically-generated bytecode.
- A working RIM4J prototype and its application to real-world software such as Apache Tomcat and ActiveMQ, as well as security and performance evaluations that confirm the effectiveness and efficiency of RIM4J.

This paper is an expanded version of our previous work [32] published in SRDS2017, and the rest of this paper is organized as follows. Section 2 reviews the necessary background on Java bytecode and integrity measurement. Section 3 illustrates the architecture overview of RIM4J and the threat model. Section 4 provides a detailed description of how the violations of bytecode integrity commonly work through the analysis of some Java exploits. Section 5 presents the implementation of the runtime measurement approach and describes the details of the dynamically-generated bytecode. The security and performance evaluations are given in Section 6. Finally, we discuss limitations, cover related work and conclude in Sections 7–9, respectively.

2. Background

In this section, we provide a basic introduction to the Java bytecode (Section 2.1). Additional, we will give an overview of integrity measurement in trusted computing and a preliminary description of how to measure a Java application within a cloud computing environment (Section 2.2).

2.1. Java Bytecode

As of 2018, Java, which is a class-based and object-oriented computer-programming language, is among the most popular languages in use [33], particularly for client-server web applications, with a reported nine million developers worldwide [34]. It is mainly attributed to the design and implementation of Java bytecode [35,36]. Java applications are typically compiled to bytecode, which can run on all platforms that support the JVM regardless of computer architecture. As shown in Figure 1, a source code file is compiled into a particular file format (ends with the .class file name extension), called ClassFile. Each class file contains one and only one class or interface.

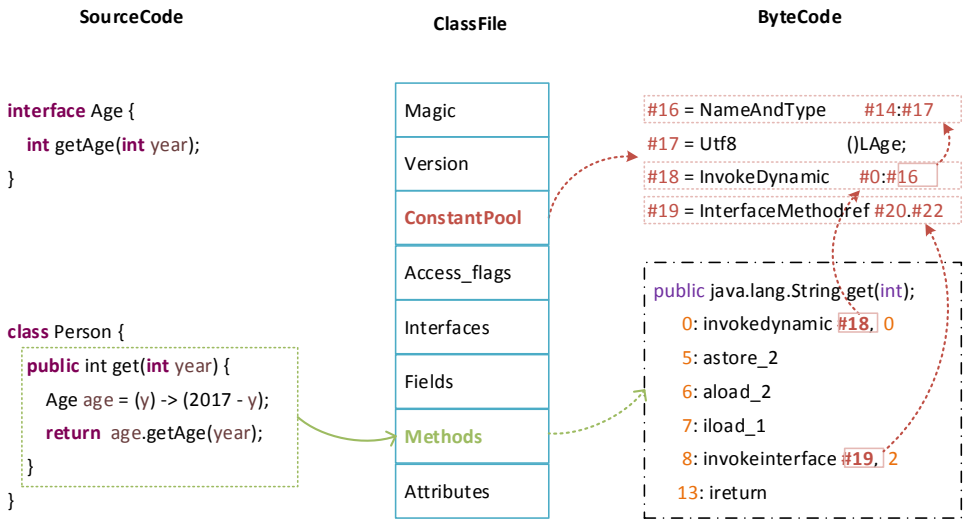

Figure 1. Relationship among source code and bytecode in the Java platform.

Figure 1 illustrates a simplified example of a Java class. It starts with a magic number (OxCAFEBABE) as a unique identifier of the JVM-supported format. The core in a class file is *ConstantPool* and *Methods*. The *ConstantPool* can be roughly thought of as the data segment of a Java application, which contains approximately 60% of an average class [37]. The *Methods* that contains the bytecode can be approximated to the text segment, which makes up 12% of an average class [37]. The remaining parts are insignificant auxiliary content, such as debug and exception information.

2.2. Integrity Measurement

The integrity of a hosted application is a binary property that indicates whether this application has been modified in an unauthorized manner. Such an unauthorized modification may lead to incorrect or malicious behaviors by the application, such that it would be unwise for a cloud customer to rely on it. Many previous efforts focused on measuring code and association integrity semantics with the code [25,38,39]. Taking an integrity measurement of code means computing a cryptographic hash (e.g., SHA-1 hash) of it and extending a hash chain with that hash value.

In the Java world, a series of classes constitutes a Java application. The *ConstantPool* and *Methods* are the core of a Java class, just as the code is for a native application. As the RIM4J measurement is taken at runtime rather than load-time, it computes a hash value of the entire measurement list instead of the extended value.

RIM4J measurement generates two values: (1) a measurement list of a Java application M containing each measurement $m_0, m_1, \cdots, m_i$ where m_i contains a hash value of a Java class h_c and $h_c = hash(ConstantPool||Methods)$; and (2) a hash aggregate H where $H = hash(m_0||m_1||\cdots||m_i)$. The attesting party provides M and a signed H to a cloud customer (the customer must be able to reliably obtain the public key certificate for its attestation identity key), and the customer verifies that a hash aggregate computed from the measurements in M corresponds to the signed hash aggregate.

3. The RIM4J Architecture

The primary goal of RIM4J is to provide runtime integrity measurements of Java applications for cloud customers, even facing outside attacks through one or more Java vulnerability exploits. The secondary goal of RIM4J is to make the approach of RIM4J practical, and thus, it can be deployed for large software systems with small overhead.

3.1. Overview

vTPM-based attestation: RIM4J requires not only integrity measurement, but also remote attestation in improving trustworthiness within cloud computing environments. Remote attestation needs the support of cryptographic engines, which is embedded physically in the hardware Trusted Platform Module (TPM). However, a physical TPM cannot be directly used by the VMs within a cloud environment; thus vTPM is designed to offer the same usage model and services to the VMs [40]. In this paper, RIM4J leverages a virtual TPM (vTPM) module to carry out directly an attestation between the cloud customers and their leased virtual machines through the vTPM instances. Specifically, RIM4J signs the integrity information through the Attestation Identity Key (AIK) of vTPM to provide the integrity and unforgeability of this information, as well as leverages a secure connection to protect the authenticity and confidentiality of an attestation.

IMA-enforced protection: Since RIM4J aims at providing runtime measurements for Java applications, it requires the underlying system to be trustworthy, which can be easily achieved with existing approaches (e.g., IMA [25]). RIM4J leverages IMA to build upon the initial trust in a guest VM by verifying the integrity of the software loaded into this VM, in the order that they are booted. Especially, RIM4J can prevent the underlying JVM from malicious adversaries by verifying the integrity of its binary content (executable files, dynamical libraries, and so on) with the correct pre-calculated reference. On the other hand, RIM4J relies on the enforcement of such W$\oplus$X mapping to protect the guest user memory, which is proven to be an effective and efficient approach to ensure runtime code integrity.

Architecture overview: Figure 2 shows an overview of the RIM4J architecture. A Java application whose "executable file" consists of classes (normal loaded class) is loaded into the memory through Java classloaders, and the JVM creates quite a few classes (dynamically-generated class) during runtime due to the implementation of language features (Section 5.2). Existing approaches [25,28] that rely on the system call interception or virtual machine introspection can measure the normal loaded classes, but not the dynamically-generated classes which are data directly created in a heap by the JVM. The incapacity in this type of measurement is due to the semantic gap in how to accurately extract the semantic meaning about the dynamically-generated classes from the outside view of the JVM. Note that there is no clear distinction between a normal loaded class and a dynamically-generated class from the JVM's perspective.

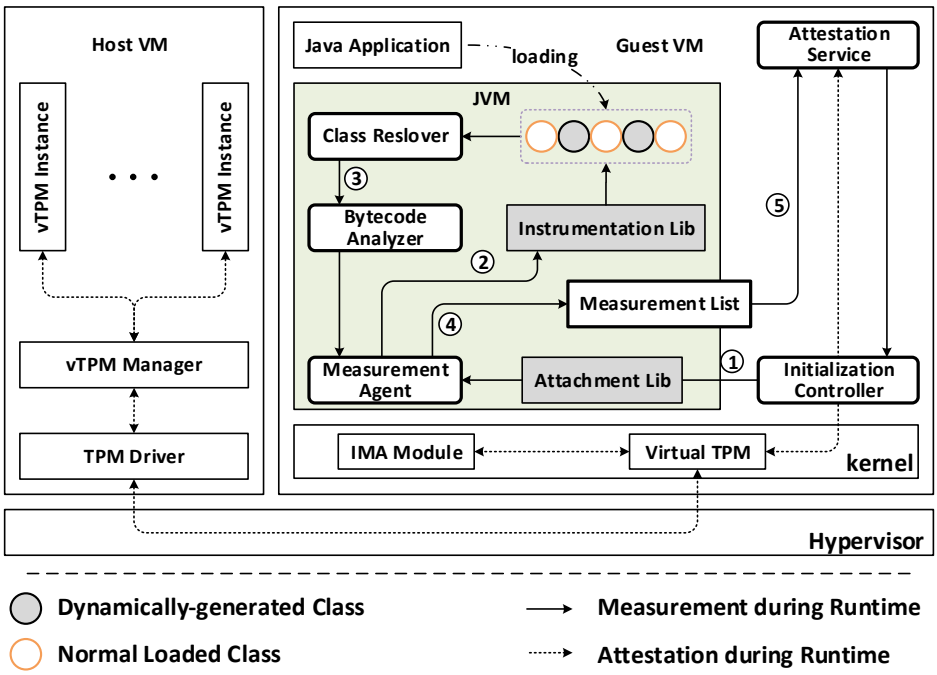

Figure 2. Architecture overview of RIM4J.

Once a cloud customer issues an attestation request, the Attestation Serviceshall then notify the Initialization Controllerto start up a Measurement Agentthrough the invocation of the attach API in the Attachment lib (Step ①). Although the Measurement Agent is running inside the same world of the Java application, the trustworthy JVM provides the isolation between the agent and the classes. Furthermore, this agent becomes online when, and only when carrying out an integrity measurement. To bridge the semantic gap, the Measurement Agent relies on the use of Instrumentation libto access the heap area storing the Java classes (Step ②). Due to the limitation of the Instrumentation lib, RIM4J only gains the entire byte content of a class, and this class is not the same as its original version at load time, which results from the JVM's performance optimization for the efficient execution of a class. To address the issues as mentioned above, RIM4J resolves and analyzes a class to extract the bytecode (Step ③), which can be similar to the text segment of a binary executable file (Section 2.1). After completing the current measurement, the agent writes measurements into a protected measurement list (Step ④), which can be retrieved by the Attestation Service (Step ⑤). During runtime, the measurement list is confined to be written by the Measurement Agent and read by the Attestation Service through the SELinuxsystem.

Measurement mode: RIM4J allows customers to invoke the measurement and attestation requests at any time during the lifecycle of the hosted applications. It also provides the customers with two modes of operation: one-time measurement and periodic measurement.

One-time measurement: The cloud customer can request the attestation at any time. Whenever receiving a request, the Measurement Agentcarries out a runtime measurement of the attested application and generates an integrity proof. Then, the Attestation Service performs the sign operation of the proof by the AIK and sends back the result to the customer.

Periodic measurement: The cloud customer can specify the frequency of attestations, specified as constant or random. The Measurement Agent enforces measurements under the specified frequency, and the Attestation Service supplies the measurements periodically. The customer receives recent fresh results and can stop the process at any time.

3.2. Threat Model and Assumptions

RIM4J aims at measuring Java applications at runtime so that a cloud customer can detect suspicious behaviors, such as remote command execution or uploading malicious files. In such a way, we assume that the goal of an adversary is to obtain persistent access to an application of a cloud customer in order to steal, disrupt or even deny the data and code of a cloud customer. Under these circumstances, the adversary needs to execute malicious bytecode in the same trusted privileged domain through one or more Java exploits. Specifically, the adversary may try to use sophisticated attacks to invalidate a secure access-control policy or bypass a load-time measurement check, so that it can directly execute bytecode in the memory during runtime.

We acknowledge the possibility that an adversary can modify bytecode between the time it is attested and the time it is used. This unaddressed problem with current code-attestation technology is the time-of-attestation and time-of-use discrepancy. Even though the bytecode is correct to work at the time of attestation, an adversary could compromise the attested bytecode by the time of use.

We consider that the cloud service provider is trustworthy, a reasonable presumption for cloud customers at present. The same assumption has been adopted by many cloud security architectures [28,30,41,42]. We, therefore, view threats such as insider adversaries and collusion attacks to be out of scope. These threats are quite valuable, but are not resolved by our RIM4J architecture for now. We also assume an adversary without the ability to physically tamper with any hardware of any host machine (i.e., CPU, memory and TPM). In cloud computing environments, the security of RIM4J depends on keeping the underlying runtime systems (i.e., Hypervisor, OS and JVM) working legitimately. Therefore, we assume that the cloud service provider does not purposefully deploy malicious underlying systems with an explicit capability to control and disrupt a hosted application of a cloud customer.

4. Java Exploits Analysis

The Java platform enables cloud customers to keep their applications away from memory corruption attacks, such as the control-flow hijack attack, but customers still suffer from the bytecode corruption attack within cloud computing environments. The bytecode corruption attack may originate from the sandbox and classloader. The Sandboxin the Java platform is supposed to enforce security restrictions to contain the execution of untrusted code. Classloaders are supposed to ensure that all code is only able to load classes that it is allowed to access. However, we find that a set of exploits can bypass sandbox restrictions that should be enforced during runtime and load classes that should be incapable of being loaded. Even worse, these exploits do not rely on any specific CPU or operating systems to work on any platform (e.g., Windows, Linux). Although the introduction of *sun.misc.Unsafe* enables cloud customers to write high-performance "systems-level" code in Java, the operations the Java's unsafe capability provides can be dangerous, as they allow one to circumvent the safety guarantees provided by the Java language and the JVM. If misused, the consequence can be bytecode corruption, meaning that an adversary loads and executes arbitrary classes without any security checks.

First, the Java sandbox model affords so much flexibility that it leads to unnecessary vulnerabilities and bad security practices [6]. In the Java sandbox model, the JVM leverages a number of security permissions to restrict the behaviors of untrusted code for securing a benign application. For cloud customers, they do not need all of the security permissions. Some permissions can yet decrease the security of the hosted application without obviously improving practical functionality. A security permission, *RuntimePermission"createClassLoader"*, enables defining arbitrary classes in the same protection domain as trusted code. Once a cloud customer unintentionally grants his/her application with this permission, it can pose a risk to the security of the application and incur significant losses. For example, in the CVE-2012-0507 [10], an adversary can invoke a classloader to create some self-defined classloader by using type confusion attacks. As depicted in Listing 1, the elaborately-crafted classloader defines an additional malicious class in an escalated privilege context, meaning that this class can perform any operations and access any resources with the intended malicious purposes.

Second, many loading flaws [7,8] in Java core class libraries and third-party middlewares have been exposed in recent years, which breach restrictions from the sandbox to execute arbitrary bytecode. Classloaders play a central role in guaranteeing the safety property of the Java platform, and they are supposed to prevent adversaries from loading untrusted classes into the same protection domain as good classes. However, recent exploits indicate that classloaders may allow an attacker to profit from a confused deputy attack or abuse security vulnerabilities in trusted classes to implement the loading of arbitrary classes. For Java class libraries, Listing 3 and 4 give various examples of the implementation to load arbitrary classes. Listing 3 uses a class *MethodHandle* to invoke arbitrary methods without the stack-based access control; Listing 4 leverages the weakness of *JmxMBeanServer* and *MBeanInstantiator* to execute any class that they should be incapable of loading. For third-party middlewares, there is a significant number of malicious exploits [13–23]. For example, CVE-2016-3088 in ActiveMQ [22] and CVE-2013-4444 [21] in Tomcat are exposed allowing some of their implementation defects to upload malicious class files and trigger their execution to be exploited. In some cases, this is due to the incapability of the self-defined classloaders implemented by third-party middlewares, which could not provide a mechanism to differentiate between benign code and malicious code.

```
public Class<?> getClass(selfClassLoader scl,
      String name, byte[] buf) {
   URL url = new URL("file:///xxxx");
   Certificate[] certs = new Certificate[0];
   Permissions perm = new Permissions();
   perm.add(new AllPermission());
   ProtectionDomain pd = new ProtectionDomain(new
      CodeSource(url, certs), perm);
   Class<?> cls = scl.defineClass(name, buf, 0,
      buf.length, pd)
   return cls;
}
```

Listing 1: Modified excerpt for CVE-2012-0507.

```
public Class<?> getClass(byte[] classFile){
/*Field theUnsafe = Unsafe.class.
      getDeclaredField("theUnsafe");
   theUnsafe.setAccessible(true);
   Unsafe unsafe = (Unsafe) theUnsafe.get(null);
*/
   sun.misc.Unsafe unsafe = sun.misc.Unsafe.
      getUnsafe();
   Class<?> cls = unsafe.defineClass(null, classFile, 0,
      classFile.length, null, null);
   return cls;
}
```

Listing 2: Modified excerpt for CVE-2012-5076.

```
public Class<?> getClass(String name) {
   MethodHandles.Lookup lookup = MethodHandles.
      publicLookup();
   MethodType methodType = MethodType.methodType(
      Class.class, String.class);
   MethodHandle methodHandle = lookup.findStatic(
      Class.class, "forName", methodType);
   Class<?> cls = (Class<?>) methodHandle.
      invokeWithArguments(new Object[] { name });
   return cls;
}
```

Listing 3: Modified excerpt for CVE-2012-5088.

```
public Class<?> getClass(String name) {
   JmxMBeanServer beanServer = (JmxMBeanServer)
      new JmxMBeanServerBuilder().newMBeanServer("
      ", null, null);
   MBeanInstantiator beanInstantiator = beanServer
      .getMBeanInstantiator();
   ClassLoader classLoader = null;
   Class<?> cls = beanInstantiator.findClass(name,
      classLoader);
   return cls;
}
```

Listing 4: Modified excerpt for CVE–2013-0422.

Last but not least, the improper use of the powerful *sun.misc.Unsafe* API is highlighted as a risky feature to circumvent Java's safety guarantees [9]. This API was originally introduced to access low-level, unsafe features of the JVM and underlying hardware, which are unavailable in safe Java bytecode. As described in [9], the *Unsafe* operation enables third-party frameworks and libraries to achieve better performance or implement functionality not otherwise available in the Java language. As the operation *Unsafe* is too low-level, it also allows undermining the safety guarantees provided by the Java language and the JVM. With the use of this powerful API, an adversary can directly execute its untrusted bytecode without any classloader and any check of security permissions. The obvious example is CVE-2012-5076 [24], in which the adversary loads the malicious class into the memory through the *defineClass* method of the *Unsafe*, as shown in Listing 2. Note that the use of the *getUnsafe* method, which assists with gaining an instance of the *Unsafe*, is limited in the Java core class libraries. However, it is possible to bypass such a restriction through reflection, as depicted in the comments of Listing 2.

5. Runtime Integrity Measurement

5.1. Java Bytecode Runtime Measurement

As noted in Section 2.1, a series of classes or interfaces constitutes a Java application. A class or interface that has a binary representation (a class file) is loaded into the memory through a built-in or self-defined classloader. A Java platform supports a set of built-in classloaders, namely BootStrap ClassLoader, Extension ClassLoader and System ClassLoader. Furthermore, it provides enough flexibilities for third-party middlewares to design their classloaders with a particular purpose. For example, Tomcat developers created the WebappX classloader to isolate web applications. As an interface is a more abstract class, we do not distinguish interfaces from classes, meaning that we view an interface as a class for simplicity in this paper. In the Java world, everything is represented as an object, even though an array or a primitive type. Note that an array or a primitive type does not have an external binary representation, and a JVM instance directly creates and associates it with the Bootstrap ClassLoader to meet the object-oriented principle.

Figure 3 shows the layout of classloaders and managed classes in the memory of JVM. A JVM arranges classloaders into a chain that originates from BootStrap Classloader and associates a class with a classloader, which is responsible for the loading and isolation of this class. Note that all classloaders except the BootStrap Classloader are implemented as Java classes and need to be measured. The BootStrap Classloader is comprised of platform-specific machine instructions that kick off the whole classloading process. This classloader loads the very first Java classloader to start process. It also is in charge of loading all of the bytecode required to support the basic Java runtime, including classes in the java.util and the java.lang packages.

From the viewpoint of JVM specification [36], a class is identified by a fully-qualified name, *PackName.ClsName*. Notations *PackName* and *ClsName* denote the name of a package and the name of a class, respectively. A class is determined not by its name alone, but by a pair: the name and the associated classloader. The same class loaded by distinct classloaders is allowed to be executed inside a virtual machine. If a class C in the package p is loaded by a classloader L_1, the class is keyed as <$p.C$, L_1> in a JVM instance. This means that two classes <$p.C$, L_1> and <$p.C$, L_2> are not the same.

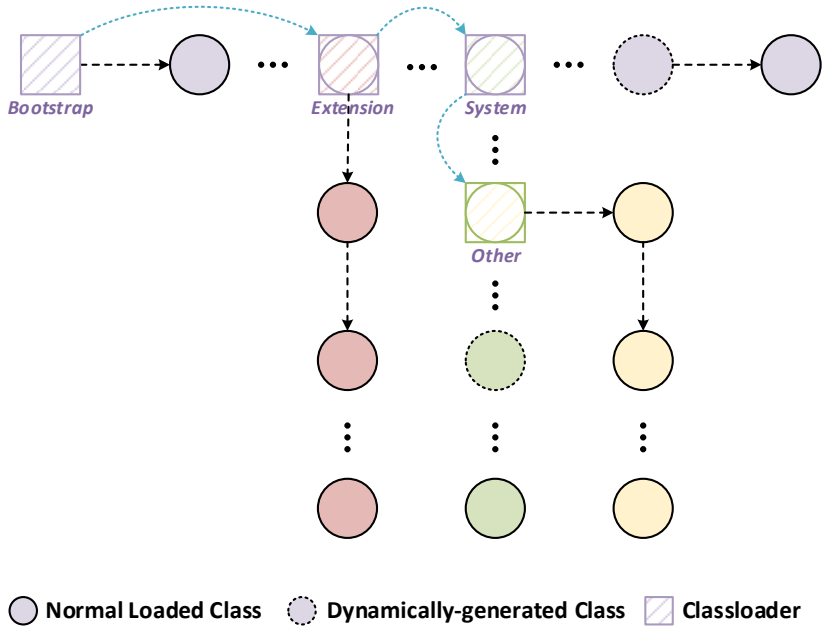

Figure 3. Classloaders and managed classes layout.

In this paper, we enforce a runtime integrity measurement of a Java application at class-level granularity. As noted in Section 2.1, the core part of a class is comprised of *ConstantPool* and *Methods*. Making a measurement of a class means computing a hash value of the *ConstantPool* and *Methods*. The runtime measurement of Java bytecode is the job of the Measurement Agent in Figure 2. The agent is designed as a plugin to achieve portability, which means that the agent only brings into full play its measuring role inside a target JVM until an integrity measurement occurs. When online, the agent would first traverse every alive classloader from the Bootstrap ClassLoader through which we can obtain all in-memory classes. The algorithm for this operation is shown in Algorithm 1 with the input of the classloader root (Bootstrap ClassLoader).

Algorithm 1: Bytecode runtime measurement.

Input : Classloader root L_r
Output: Runtime measurement list M_r

```
for every alive classloader reached from the root L_r do
    traverse this classloader to obtain relevant in-memory classes;
    store in a set of classes C_s;
end
for each class ∈ C_s do
    if class is not Primitiveor Array then
        parse class to extract bytecode bc;
        measure bytecode bc to generate an integrity hv;
        gain class name cn and loader name ln from class;
        add a tuple <cn, ln, hv> into M_r;
    end
end
```

On account of the fact that a class is only associated with a classloader, all in-memory classes are found by the traversal of alive classloaders and stored into the C_s. Due to the object-oriented principle, the JVM also represents primitive data types and arrays as classes, but they do not have any executive logic, just an abstract representation. As a result, it is necessary to exclude them from a set of classes to

be measured. The detailed measurement procedure refers to these operations in Lines 6–9. In order to compute a hash value as an integrity value of a class, the first thing to do is to extract bytecode from *class* using our efficient parsing scheme. When obtaining bytecode *bc* and generating a value *hv*, we also need the class name *cn* and the classloader name *ln* as a unique identifier to constitute a tuple <*cn*, *ln*, *hv*> and to be added into a measurement list M_r.

Note that the time complexity of the algorithm is $O(n_c)$ where n_c represents the number of the loaded classes (the length of the set C_s). During the analyses of the algorithm, we mostly consider the worst case scenario, i.e., when *class* is not a primitive type or an array. In this worst case, the *If* condition will run n_c times. In addition, the first *For* will run n_l times where n_l is the number of alive classloaders. Therefore, the total execution time T will be $O(n_c + n_l)$ where we use O-notation to denote the asymptotic upper bound. However, we find that in practice, the number of classloaders is far less than the number of classes ($n_l \ll n_c$). We then ignore the lower term and let $T = O(n_c + n_l) = O(n_c)$. We can see clearly that the total time linearly depends on the length of the set C_s. If the length of the set were to increase, the execution time of the algorithm would also increase.

5.2. Dynamically-Generated Bytecode

With the ongoing programming-language evolution, new techniques and mechanisms are disseminated in the Java landscape, such as Java Proxy, CGLIB, annotation, reflective invocation and lambda expressions. They are implemented using dynamically-generated bytecode at runtime.

When we want to add or modify some functionality of an already existing class, we can dynamically create a proxy class instead of the original delegating one through Java Proxy or CGLIB at runtime. However, there are some differences between Java Proxy and CGLIB. The Java Proxy requires that the delegating class implement any set of interfaces; however, the CGLIB extends the original class without the *final* modifier.

Annotations are metadata for the code to provide information about the attribute (class/method/package/field) on which it is defined. However, an annotation is a particular kind of interface type, meaning that annotation does not contain any executive logic. Therefore, it needs to create an instance of obtaining attribute information, which is precisely implemented by using Java Proxy.

It is true that the Java Reflection API comes with a perceptible runtime overhead, which is mainly called forth by the lookup of a method. However, if calling this method often enough, the JVM will take care of genuinely optimizing these reflective invocations. This concept is labeled inflation and is incidentally implemented by code generation at runtime.

Lambda expressions are designed to dramatically raise abstraction to make Java code more generic, flexible and reusable. Lambda expressions can only appear in places where they will be assigned to a variable, and the type of variable is a functional interface. JVM developers came up with an approach to implement lambda expression so as to maximize flexibility for future optimization and provide stability in the class file representation. It translates lambda expression into the bytecode until runtime by the use of *invokedynamic*. When invoked, the JVM generates an instance of the functional interface to which the lambda expression is converted.

The use of dynamically-generated bytecode promotes the implementation of these above mechanisms in the Java world. From the view of a JVM instance, there does not yet seem to be much of a difference between a normal loaded class and a generated class. However, these generated classes have a set of features in a class name (shown in Table 1) or may be associated with a class loader. The features can be leveraged to differentiate dynamically-generated classes from normal loaded classes. For example, the name of a class generated from Java Proxy contains *$Proxy*; a new *Delegating ClassLoader* defines a class generated due to the reflective invocation.

The strength of the protection provided by RIM4J is determined by the measurement it enforces. The accuracy of a measurement approach is determined by the relationship between false negatives and false positives. The possibility of protection failure (false negatives) relies upon the scope of the measurement. It is possible for an adversary to implement its malicious intention in the generated

bytecode. Prior works [43,44] do not measure dynamically-generated bytecode, leading to high false negatives. The avoidance of false alarms (false positives) is a stringent requirement for any practical solution. Faults in normal operation are unacceptable in production environments. Consequently, it is necessary to identify the generated bytecode and make a more accurate analysis for subsequent verification. It is for this reason that we take the dynamically-generated bytecode into the scope of the measurement. We expect that enforcing this type of measurement could discover legal generated bytecode to restrain protection failure situations and unnecessary incorrect operation.

Table 1. A dynamically-generated class name feature.

Mechanism	Class Name Feature
Java Proxy	package name of a non-public proxy interface + "." + "*$Proxy*" + num "com.sun.proxy" + "." + "*$Proxy*" + num
CGLIB	delegating-class name + "$$" + class-generator name + "By*CGLIB$$*" + num
Annotation	package name of a non-public proxy interface + "." + "*$Proxy*" + num "com.sun.proxy" + "." + "*$Proxy*" + num
Reflective Invocation	"*sun.reflect.GeneratedSerializationConstructorAccessor*" + num "*sun.reflect.GeneratedConstructorAccessor*" + num "*sun.reflect.GeneratedMethodAccessor*" + num
Lambda expressions	calling-class name + "*$$Lambda$*" + num

6. Evaluation

In this section, we evaluate RIM4J by examining its effectiveness, as well as measuring both its performance and other aspects. In this section, we first describe our experimental setup, followed by effectiveness evaluation, performance study and portability evaluation.

6.1. Experimentation Setup

Our testbed is a Lenovo ThinkPad T440s, featuring a 1.7-GHz Intel (R) Core i5-4210U CPU, 12 GB of RAM with Xen 4.10. We integrated the TPM-emulator and leveraged it to emulate the functions of the trusted module in the hardware. The Host VM is the privileged domain of the Xen virtualization platform with a Linux Mint 18.3 OS (4.8.0 Linux kernel). The Guest VM is used to deploy Java applications by a tenant, running CentOS 7.4 OS (3.10.0 Linux kernel).

6.2. Effectiveness Evaluation

We have used RIM4J to detect a set of Java shellcodes for known CVE vulnerabilities, classifying them into three categories: excess sandbox permission, Java loading flaws in core class libraries and third-party middlewares and *sun.misc.Unsafe* API abuse, as shown in Table 2. For excess sandbox permission, a shellcode often relies on *RuntimePermission "createClassLoader"* to create some self-defined controlled classloader. For Java loading flaws, either Java libraries or third-party middlewares manipulate existing classloaders to execute arbitrary bytecode. For *sun.misc.Unsafe* API abuse, it is hard to distinguish between benign core classes and bad ones, which are loaded by the same primordial Bootstrap ClassLoader. In this case, RIM4J needs to not only compare two different views of classloaders, but also perform an integrity measurement and verify them.

For brevity, we show the detailed experiment about Apache ActiveMQ's CVE-2016-3088 [22] in the following part of this section. We construct this experiment using the shellcode to test our system's capability to detect possible attacks. First, we start with a perfectly trusted state of ActiveMQ middleware, which is also a Java application, and enforce the bytecode measurement at runtime. Second, we launch an attack by exploiting security breaches in ActiveMQ's HTTP put and move implementation. We then upload malicious bytecode (a web shell for executing commands in the parameter of an HTTP request). After that, we take another measurement.

Table 2. Category of Java exploits to execute arbitrary bytecode.

CVE ID	Software	Category
CVE-2018-1000146	Jenkins	Java Loading Flaws
CVE-2017-12617	Apache Tomcat	Java Loading Flaws
CVE-2017-12615	Apache Tomcat	Java Loading Flaws
CVE-2016-3088	Apache ActiveMQ	Java Loading Flaws
CVE-2014-0116	Apache Struts	Java Loading Flaws
CVE-2014-0114	Apache Commons BeanUtils	Java Loading Flaws
CVE-2014-0113	Apache Struts	Java Loading Flaws
CVE-2014-0112	Apache Struts	Java Loading Flaws
CVE-2014-0094	Apache Struts	Java Loading Flaws
CVE-2013-4444	Apache Tomcat	Java Loading Flaws
CVE-2013-0422	Oracle Java	Unsafe API Abuse
CVE-2012-5088	Oracle Java	Java Loading Flaws
CVE-2012-5076	Oracle Java	Java Loading Flaws
CVE-2012-0507	Oracle Java	Excess sandbox permission
CVE-2010-1622	Spring Framework	Java Loading Flaws

Figure 4a lists partial measurements of the good application, and Figure 4b gives the corresponding measurement list of the same application that is compromised by a web attack. The italicized entries imply that there are extra classes after carrying out an attack. This example illustrates how to detect such attacks by RIM4J successfully. Note that the second italicized entry states clearly that a command from a remote adversary was executed, which is implemented by the dynamically-generated bytecode during runtime. This is the reason why we must take runtime measurements of this generated bytecode. Other regular entries show that the classes could be defined by various class loaders, some of which were implemented by third-party middlewares, such as java.net.URLClassLoaderand WebAppClassLoader. From this point of view, it is a better option to carry out bytecode measurements at runtime rather than load-time to ensure the accuracy of our measurement system.

```
...
com.sun.proxy.$Proxy0                        BootstrapClassLoader     E2CFC0DF7564CEACCB970F0356EEECCE1412FB2F
org.apache.activemq.console.Main             AppClassLoader           66C377EC9D584B142B2037412 6399B3C0ED1D191
sun.reflect.GeneratedMethodAccessor7         DelegatingClassLoader    88DFE3874561F5070F69F10725BE03C3D1E95837
com.sun.proxy.$Proxy5                        java.net.URLClassLoader  E18A3630350239D882CA69A31B8DCBC02DCDEBEF
org.eclipse.jetty.webapp.WebAppClassLoader   java.net.URLClassLoader  52C756A759210EC9619291840225E09479AC0190
org.apache.activemq.web.WebConsoleStarter    WebAppClassLoader        F2D7EECC74DCF981A338E9723DD59EFC3E8CB0C0
...
```

(**a**)

```
...
com.sun.proxy.$Proxy0                        BootstrapClassLoader     E2CFC0DF7564CEACCB970F0356EEECCE1412FB2F
org.apache.activemq.console.Main             AppClassLoader           66C377EC9D584B142B2037412 6399B3C0ED1D191
sun.reflect.GeneratedMethodAccessor7         DelegatingClassLoader    88DFE3874561F5070F69F10725BE03C3D1E95837
com.sun.proxy.$Proxy5                        java.net.URLClassLoader  E18A3630350239D882CA69A31B8DCBC02DCDEBEF
org.eclipse.jetty.webapp.WebAppClassLoader   java.net.URLClassLoader  52C756A759210EC9619291840225E09479AC0190
org.apache.activemq.web.WebConsoleStarter    WebAppClassLoader        F2D7EECC74DCF981A338E9723DD59EFC3E8CB0C0
...
org.apache.jsp.test_jsp                      JasperLoader             7F07A52153F3824A8622FE0DB60A7F91E667B14C
...
java.lang.UNIXProcess$$Lambda$9              BootstrapClassLoader     7BCF5DF4504917E2CDD610C6C59295EF259C6FA2
...
```

(**b**)

Figure 4. Measurement list example consisting of the class name, class loader, and SHA-1 hash value. (**a**) Measurement example; (**b**) measurement after installing malicious bytecode.

6.3. Performance Evaluation

While our architecture reduces the risks of security-relevant malicious exploitations, these changes have a performance overhead cost. We evaluated RIM4J in the dimensions of performance (as measured by runtime overhead and memory overhead). We have also compared the bytecode resolving overhead

of RIM4J with that of DTEM [43,44]. We were restricted from comparing against other bytecode measurement systems, as many were unavailable for downloading and did not use standardized benchmarks in their evaluations. For all experiments, no other applications were running, and the system was otherwise at rest. For the macro- and micro-benchmarks, we used two JVMs: Oracle HotSpot JVM, Version 1.8.0_151, and the OpenJDK IcedTea JVM, of the same version.

6.3.1. Bytecode Measurement Overhead

Our first performance evaluation focuses on the measurement overhead. This overhead contains the bytecode resolving and measurement overhead. As shown in Table 3, we utilized several popular open source applications to evaluate the performance overhead.

Table 3. Description of the popular open source application workloads used in our experiments.

Workload	Version	Description
Cassandra	3.11.1	Distributed NoSQL database system
Hadoop	3.0.0	Distributed storage and processing framework
Maven	3.5.2	Build automation tool used for Java projects
Lucene	7.2.1	Information retrieval software library
Tomcat	9.0.4	Http web server with Java Servlet Container
ActiveMQ	5.15.3	Message broker together with a full JMSclient
Netty	4.1.21	Non-blocking I/O client-server framework
XWiki	9.5	Wiki software platform with extensibility

First, we analyzed and resolved these applications to gain the resolving time separately using the bytecode resolving algorithm of DTEM and JCLOUDATT in the same test environment. This procedure was repeated 20 times, starting a new process to run each experiment, and we then averaged these results.

Figure 5 shows the normalized resolving overhead in each bytecode measurement architecture. The figure shows that our RIM4J achieves the lowest resolving overhead in these real-world applications in comparison to DTEM. Compared with RIM4J, the average resolving overhead of DTEM is 17× that of RIM4J for `Hadoop` and `XWiki` in the best case, but 35× for `Lucene` in the worst case.

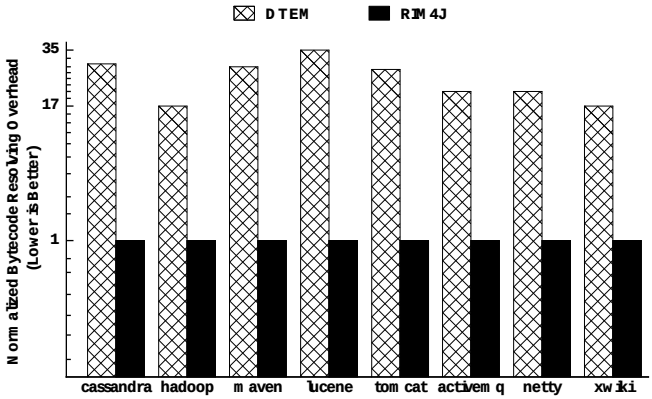

Figure 5. Bytecode resolving overhead comparison between DTEM [43,44] and RIM4J (the Y-axis is the overhead normalized to RIM4J).

We also used the same eight applications described above to evaluate the runtime overhead and memory overhead when introducing our bytecode measurement approach. To compare broadly to the measurement of the entire class (named Class Measurement), we computed the runtime overhead of bytecode measurement in various hash algorithms (SHA-1, SHA-256, SHA-512), respectively. Figure 6 shows the relative runtime overhead of bytecode measurement in each hash algorithm in comparison

with the measurement of the entire class. We describe the relative memory overhead of bytecode measurement in Figure 7. From the two figures, we make three observations.

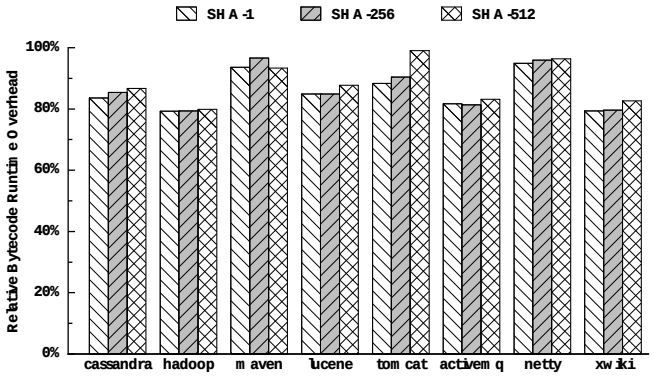

Figure 6. Runtime overhead with various hash algorithms (SHA-1, SHA-256, and SHA-512) in the bytecode measurement (the Y-axis is the runtime overhead of the bytecode measurement relative to the measurement for the entire class).

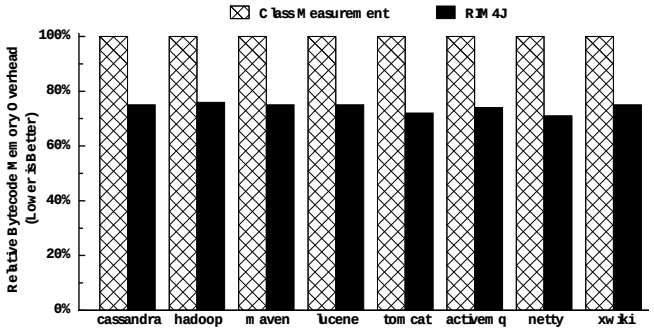

Figure 7. Memory overhead in the bytecode measurement based on the SHA-1 algorithm (the Y-axis is the memory overhead of the bytecode measurement relative to the measurement for the entire class).

(1) From Figure 6, we see the runtime overhead of our bytecode measurement approach approximately reduced from 1% to 21% in various hash algorithms. We see the best case for `Hadoop`, reduced by about 21% in each hash algorithm, but the worst case for `Tomcat`, reduced by about 1% in the SHA-512 algorithm. From our inspection, `Tomcat` contains 3325 classes, one of the least, and `Hadoop` has 130,835 classes (the most). The time taken by hash algorithms is actually in proportion to the space of the measured applications. The more classes an application has, the more space it occupies. We believe that this is the prime reason for the above observation.

(2) As depicted in Figure 7, the memory overhead of bytecode measurement is reduced by at least 24% in comparison with the class measurement. Figure 7 shows the best in `Netty` and the worse in `Hadoop`; however, there is little difference between the two cases, roughly 5%. As presented above, the bytecode (*ConstantPool* and *Methods*) makes up approximately 72% of a class on average. As our observation conforms to the actual statistic, our bytecode measurement approach has less memory overhead.

(3) Figure 6 shows that the relative runtime overhead is higher with the increased intensity of hash algorithms, except for the `Maven`. In the SHA-256 algorithm, it has higher overhead than the SHA-512. The discrepancy results from nearly the same runtime overhead between the two types of measurements of `Maven`. The exception is also attributed to a factor: `Maven` contains 6024 classes, meaning that it occupies less space to have less runtime overhead.

6.3.2. Performance Effect on Macro-Benchmarks

Our first evaluation of the performance effect focuses on macro-benchmarks, from the SPECjvm2008 [45] benchmark suite. The SPECjvm2008 benchmark suite contains 10 benchmarks that consist of a variety of common general purpose application computations designed to be representative of real-world usage. In all cases, we used the "base" category of the benchmark suite without any warm-up phase.

First, we ran the benchmarks using both the Oracle HotSpot JVM and the OpenJDK IcedTea JVM in our test environment to measure baseline execution performance (ops/m metric). Then, we started our RIM4J to initiate the Measurement Agent in both JVMs, which were executing all of the benchmarks to separately perform periodic runtime measurement at a frequency of 5 s, 10 s, 30 s and 60 s; we obtained the resulting execution performance. This process was repeated 20 times, starting a new JVM to run each experiment, and we then averaged these results.

Figure 8 shows the performance effect of our JCLOUDATT architecture in OpenJDK IcedTea 8 by varying the frequency. It gives the results of the SPECjvm2008 benchmark suite. Figure 9 shows the results in Oracle HotSpot 8. From the two figures, we make the following observations:

(1) When JCLOUDATT is turned on to enforce runtime measurement, it induces a performance degradation on the execution of a benchmark. In Figure 8, we see the lowest effect in the `scimark` benchmark and the highest effect in the `compiler` benchmark. For the `scimark` benchmark, it is down to 93% when at a frequency of 60 s. The `compiler` benchmark has a 94% reduction of performance at a frequency of 5 s. The observation is also made in Figure 9, except for the lowest: in Oracle HotSpot 8, it is down to 95% of the application performance for the `scimark` benchmark.

(2) At the same frequency, RIM4J always imposes the least reduction of application performance for the `compress` benchmark, but for the `compiler` benchmark, it yet induces the most significant reduction from Figures 8 and 9. When enforcing a measurement, JCLOUDATT needs to execute the VM operation about class redefinition to obtain the bytes of all loaded classes. This operation requires Stop-the-Worldpause for consistency. Due to a high object allocation rate [46], the `compiler` benchmark has the most significant reduction of execution performance.

(3) From Figures 8 and 9, the lower the periodic measurement frequency is, the less reduction RIM4J induces. Compared to other frequencies, each benchmark has the least reduction at a frequency of 60 s. We believe that the performance effect will be negligible if the measurement frequency is low enough.

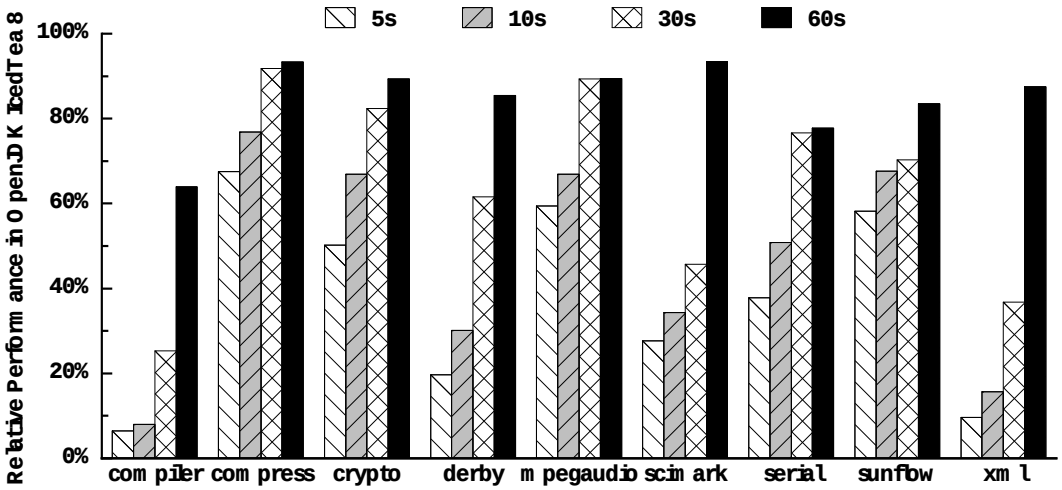

Figure 8. Performance effect of periodic runtime measurement for macro benchmarks in OpenJDK IcedTea 8 (the Y-axis is the performance relative to the baseline test; the same benchmarks, but with no measurement).

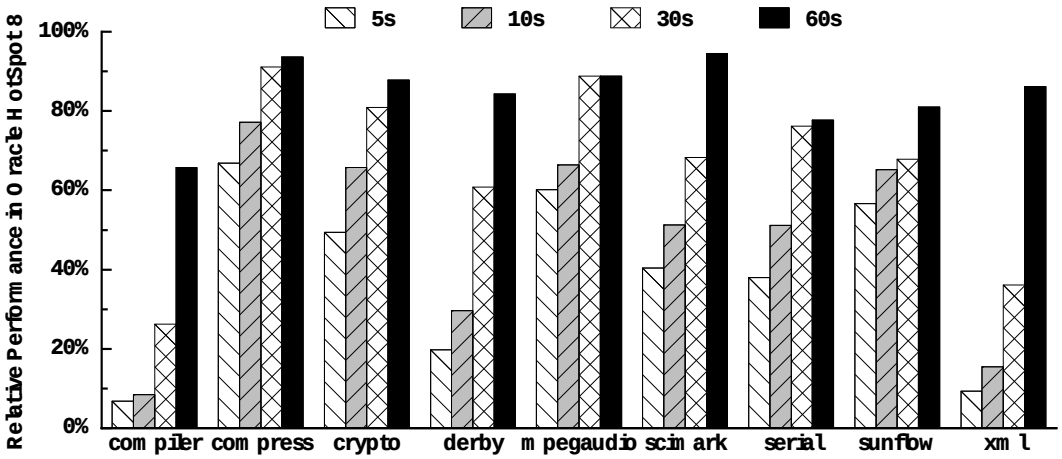

Figure 9. Performance effect of periodic runtime measurement for macro benchmarks in Oracle HotSpot 8 (the Y-axis is the performance relative to the baseline test; the same benchmarks, but with no measurement).

6.3.3. Performance Effect on Micro Benchmarks

To further analyze the runtime performance effect, we performed a series of micro benchmarks, JavaGrande [47]. The benchmark suite consists of nine benchmarks: "Arith" (execution of arithmetic operations), "Assign" (different types of variable assignment), "Cast" (casting between different primitive types), "Create" (creating objects and arrays), "Loop" (for, reverse for, while loop), "Math" (execution of math library operations), "Method" (method invocation), "Serial" (serialization) and "Exception" (exception handling).

Each benchmark was executed 20 times (each time in a separate JVM) and the results of each trial averaged. This entire process is similar to the experiment of the performance effect on macro benchmarks. In all cases, we ran the benchmark suite with no warm-up phase, as JavaGrande's benchmark runner does not support warm-up.

Figures 10 and 11 show the performance effect of our RIM4J architecture in OpenJDK IcedTea 8 and Oracle HotSpot 8, respectively, by varying the measurement frequency to 5 s, 10 s, 30 s and 60 s. From the two figures, we make the following observations:

(1) From Figure 10, we see that the lower the measurement frequency is, the lower the performance effect is. The same observation is almost made in Figure 11, as well. For each benchmark, it has the shortest reduction of performance at a frequency of 60 s except for the `Arith` benchmark in Figure 11. For the `Arith`, RIM4J imposes a 68% reduction at a frequency of 60 s, but a 60% reduction when at a frequency of 5 s. We attribute these small differences mostly to the instabilities of the JVM rather than the impact from our RIM4J [48].

(2) We see that RIM4J has a negligible performance effect on `Cast` and `Loop` in the two figures. `Cast` consists of four operations, and the result of each operation (casts/s metric) is infinity regardless of the measurement frequency. An operation of the type casting takes such a negligible amount of time that our RIM4J has little impact on this benchmark. Although the `While` in the `Loop` benchmark generates a finite result, they are very nearly equal no matter what the measurement frequency is.

(3) For `Assign`, there is little difference in performance at any frequency. We also achieve the same observation in `Method` except for the frequency of 5 s, in which it is down to 0.3% of the baseline performance in both the Oracle HotSpot JVM and the OpenJDK IcedTea JVM. In two cases, RIM4J has significantly high overhead, likely since it performs Stop-the-World pause. However, the metric of `Assign` is reduced from 1.4×10^{10} to 9.5×10^{8} and the `Method` is reduced from 7×10^{10} to 3×10^{9}, which are considered acceptable in practice.

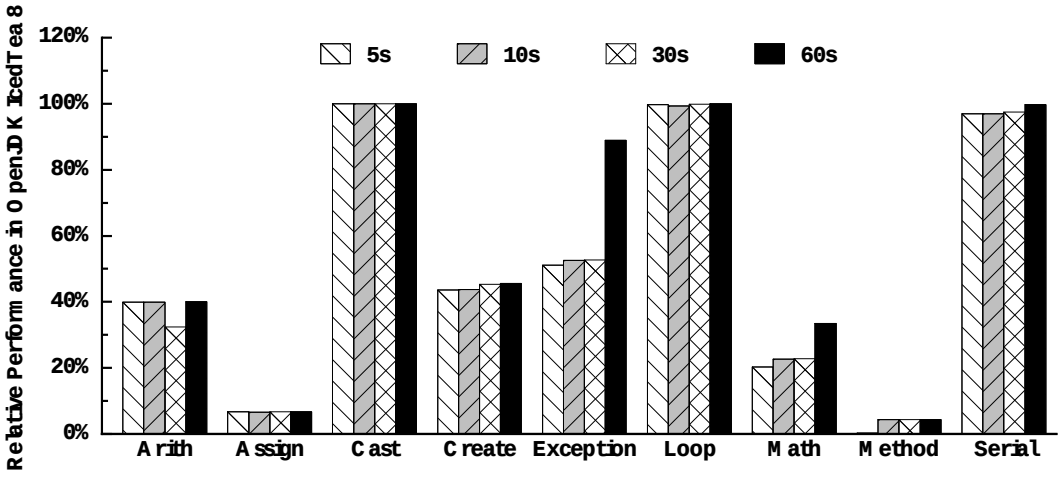

Figure 10. Performance effect of periodic runtime measurement for micro benchmarks in OpenJDK IcedTea 8 (the Y-axis is the performance relative to the baseline test; the same benchmarks, but with no measurement).

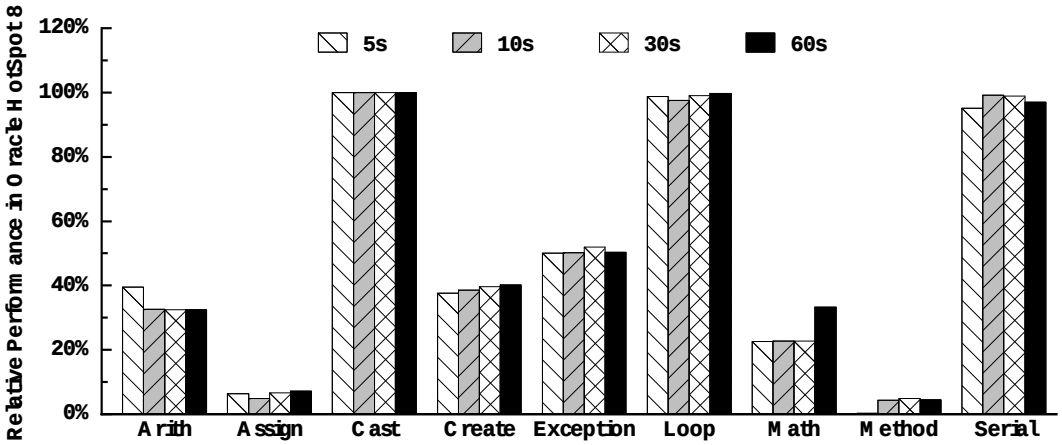

Figure 11. Performance effect of periodic runtime measurement for micro benchmarks in Oracle HotSpot 8 (the Y-axis is the performance relative to the baseline test; the same benchmarks, but with no measurement).

6.4. Portability

We further studied the portability of RIM4J by attempting to apply it to two completely different JVMs (Oracle HotSpot and OpenJDK IcedTea). For each VM, we attempted to execute two benchmark suites, the DaCapo [49] benchmark suite (9.12-MR1-bach) and the JavaGrande [47] benchmark suite (Sequential Version 2.0), as an indicator of whether RIM4J would work.

First, we use the DaCapo benchmark suite containing 14 benchmarks, which exercise popular open source applications with workloads designed to be representative of real-world usage. RIM4J passed all of these tests except for two benchmarks `batik` and `eclipse` because their execution led to errors on the OpenJDK whenever our RIM4J was turned on or off. `batik` fails with a ClassNotFoundException, and `eclipse` fails during its checksum validation.

To add additional validity to our claim that RIM4J is portable, we also utilized the JavaGrande, designed to be representative of scientific and other numerically-intensive computation. RIM4J passed all of these tests.

However, we were unable to successfully apply RIM4J to other JVMs, for example, IBM J9 VM and Jikes RVM. This is due to our inherent design limitation that our RIM4J rely on the specific API provided by Oracle's HotSpot and OpenJDK's IcedTea during the runtime measurement. We believe

that it is possible to modify RIM4J to be compatible with J9 and Jikes, but have not investigated this further.

7. Discussion and Limitation

Enforce security: Our RIM4J architecture is non-intrusive and does not protect a Java application from malicious bytecode injection attacks at runtime. However, our approach can also be retrofitted to enforce security within a cloud computing environment. In this case, when compromised or under-attack behaviors of a hosted application are detected, we can directly terminate this application to stop from further attacks and report the illegal status to the cloud customer. Note that the enforcement requirements in an attesting cloud party may not be the same as those of cloud customers, and consequently, the customers still need to perform an integrity validation for the current attestation.

Missed measuring time window: RIM4J might miss some attacks that happen between two adjacent measurements. For example, an adversary executes malicious bytecode after the current measurement, but the bytecode would automatically perform self-destruction to cover the tracks before the next. It is a fundamental limitation with the RIM4J using a periodic mechanism. Fortunately, it is rare in practice, as an adversary usually seeks persistent access to the targeted system and furthermore needs to find out the frequency in advance. Even though, it can be still detected if the RIM4J works at a random frequency.

Interference from kernel rootkits: Our architecture is restricted to measuring bytecode inside a JVM instance at runtime. It requires that the virtual machine and the underlying systems (such as hypervisorand OS) are trustworthy. As a result, kernel rootkits might interfere with RIM4J, e.g., aborting a normal measurement. In this case, we leverage the IMA module [25] at the OS level to protect the RIM4J against the rootkits. Given the strong isolation supported by the hypervisor, it may be a better option to build our architecture in an out-of-the-box way through virtual machine introspection [50].

8. Related Work

Application measurement: Integrity measurement in trusted computing is intended to generate an integrity proof of an attested target, and an attesting party would send this proof to a cloud customer to validate whether the target is trustworthy. There are numerous approaches to address integrity challenges at the application level, as depicted in Table 4.

Table 4. Qualitative comparison of integrity measurement approaches for applications. A symbol (✓) in the term "Portability" indicates that it is not needed for any modification of the underlying systems or access to the source code.

Approaches	Content		Measurement		Portability			
	Binary	Bytecode	Runtime	Periodic	Hypervisor	OS	JVM	Source Code
IMA [25]	✓	✓			✓	✓		✓
SRA [51]		✓			✓	✓	✓	
PRIMA [26]	✓				✓			✓
DR@FT [27]	✓				✓			✓
TBVMM [52]		✓			✓	✓	✓	
DTEM [43,44]		✓	✓	✓	✓	✓		✓
JMF [29]		✓	✓		✓	✓		
OB-IMA [28]	✓	✓					✓	✓
CloudMonatt [30]	✓		✓	✓		✓	✓	✓
Ta-TCS [31]	✓		✓	✓			✓	✓

Linux IMA [25] is the first to extend TCG's measurement scope to the application level. IMA intercepts some critical system calls to measure the code before being executed (load-time measurement). As the extension of IMA, PRIMA [26] is based on the CW-Lite integrity model [53]

to eliminate many needless measurements. DR@FT [27] was further proposed to describe the different trustworthiness through a ranking scheme to give much richer semantic information in an attestation. OB-IMA [28] is an out-of-the-box approach to rely on system call interception to measure security-critical files, such as the executable content of applications. CloudMonatt [30] and Ta-TCS [31] leverage Virtual Machine Introspection (VMI) [50,54] to implement a hypervisor-level monitor to examine the code integrity of an application process in a guest virtual machine (VM) at runtime.

These approaches focus primarily on the integrity of binary code, the majority of which are load-time measurements [25–28]. Practically all modern CPU and OSes support setting non-writable page permissions, which is simple and efficient to ensure code runtime integrity [5]. Unfortunately, the runtime integrity does not support bytecode, as OSes view bytecode as data, meaning that the bytecode resides in writable heap areas. Although CloudMonatt [30] and Ta-TCS [31] provide runtime measurements in cloud computing, they aim to enable code integrity rather than bytecode integrity.

RIM4J leverages the existing application measurement approach [25] and commodity hardware feature to guarantee the load-time code integrity of the JVM and enforce the W$\oplus$Xpolicy for its runtime integrity. Additionally, we extend the scope of the measurement to the level of bytecode and support the runtime measurement to detect undesirable behaviors of Java bytecode.

Bytecode measurement: Semantic Remote Attestation (SRA) [51] verifies the property integrity of high-level programs by using a high-level language virtual machine. It needs access to the source code to collect properties, and TBVMM [52] also needs this. Thober [29] describes the Java Measurement Framework (JMF) to detect an integrity violation of Java applications at runtime. However, it also requires modifications for JVM, which restricts its widespread usage. DTEM [43,44] implements load-time and runtime measurements for bytecode integrity, but it still modifies the class loadersof the Java platform.

As a bytecode measurement, RIM4J leverages modern virtual machine features to instrument the measurement agentinto the JVM dynamically at the level of the bytecode, meaning that RIM4J gains the portable capability to be easily deployed and adopted for real-world purposes.

Dynamically-generated bytecode: The measurement systems as mentioned above do not support the measurement of dynamically-generated bytecode, although it is common that JVM developers implement some favorite language features through dynamically-generated bytecode. The underlying JVM directly creates the dynamically-generated bytecode inside the native heap, which is the pre-allocated memory due to the performance optimization. As such, the approaches as mentioned above cannot perform any measurement by using system call interception or virtual machine introspection. To the best of our knowledge, RIM4J is the first measurement approach for this type of bytecode integrity.

Java security: RIM4J continues the line of research of protecting Java applications from malicious behaviors, including access-control checks [55,56], taint analysis [57,58] and Java cryptography security [59]. RIM4J adds to the literature by showing that the portable integrity measurement approach can be leveraged for the detection and verification of undesirable behaviors from malicious adversaries, such as uploading arbitrary files and remote code execution.

9. Conclusions

Memory-safety JVM provides an efficient way to secure cloud applications due to the control flow integrity. However, there is still another violation of bytecode integrity. It results primarily from the given excess sandbox permission, loading flaws in Java libraries and third-party middlewares and *sun.misc.Unsafe* API abuse. In this paper, we propose a language-supported architecture, called RIM4J, to further secure Java applications within a cloud system. RIM4J enforces a runtime measurement of Java bytecode and attests this to a cloud customer in an unforgeable manner. To the best of our knowledge, our RIM4J is the first to measure dynamically-generated bytecode. RIM4J leverages language-supported features to be entirely portable to be applied seamlessly to other platforms. We apply our runtime measurement system to a messaging server application where

we show how RIM4J can detect undesirable invocations, such as uploading arbitrary files and remote code execution. Evaluations show that RIM4J achieves as low as a 1/35-bytecode resolving overhead for DTEM (1/17 at worst). We evaluate RIM4J's performance effect on the SPECjvm2008 macro-benchmark suite and JavaGrande micro benchmark suite in two commonly-used JVMs, finding the best performance to approach 95% and 100% of the baseline, respectively.

We plan to extend our work in several directions. First, RIM4J only does periodic runtime measurement for the bytecode. We plan to extend RIM4J to enforce an active measurement for malicious bytecode discovery in a real-time manner, but with lower overhead. Second, we expect to provide a property-based attestation for other cloud customers rather than just tenants, such as end users. Third, we intend to deploy RIM4J to mobile platforms to see the performance implications.

Author Contributions: H.B. and H.Z. contributed equally to the design of the ideas, the analysis of results and the writing of the paper. H.Q., Z.W. and J.R. proofread the paper.

Funding: This research is supported by the National Natural Science Foundation of China (Grant No. 61303191 and No. 61402508) and the National High Technology Research and Development Program of China (Grant No. 2015AA016010).

Acknowledgments: We would like to acknowledge our families, friends and colleagues for their support, suggestions and reviews while conducting this study. We are also thankful to them for the successful completion of this part of the project.

Conflicts of Interest: The authors declare that there is no conflict of interests regarding the publication of this manuscript.

Abbreviations

The following abbreviations are used in this manuscript:

JVM	Java Virtual Machine
OS	Operating System
VM	Virtual Machine
VMI	Virtual Machine Introspection
SRA	Semantic Remote Attestation
TPM	Trusted Platform Module
AIK	Attestation Identity Key
vTPM	virtual TPM
IMA	Integrity Measurement Architecture

References

1. Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A.D.; Katz, R.; Konwinski, A.; Lee, G.; Patterson, D.; Rabkin, A.; Stoica, I.; et al. A View of Cloud Computing. *Commun. ACM* **2010**, *53*, 50–58. [CrossRef]
2. Singh, A.; Chatterjee, K. Cloud security issues and challenges: A survey. *J. Netw. Comput. Appl.* **2017**, *79*, 88–115. [CrossRef]
3. Khan, M.A. A survey of security issues for cloud computing. *J. Netw. Comput. Appl.* **2016**, *71*, 11–29. [CrossRef]
4. Singh, S.; Jeong, Y.S.; Park, J.H. A survey on cloud computing security: Issues, threats, and solutions. *J. Netw. Comput. Appl.* **2016**, *75*, 200–222. [CrossRef]
5. Szekeres, L.; Payer, M.; Wei, T.; Song, D. SoK: Eternal War in Memory. In Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP'13), Berkeley, CA, USA, 19–22 May 2013; pp. 48–62.
6. Coker, Z.; Maass, M.; Ding, T.; Le Goues, C.; Sunshine, J. Evaluating the Flexibility of the Java Sandbox. In Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15), Los Angeles, CA, USA, 7–11 December 2015; pp. 1–10.
7. Garber, L. Have Java's Security Issues Gotten out of Hand? *Computer* **2012**, *45*, 18–21. [CrossRef]
8. Holzinger, P.; Triller, S.; Bartel, A.; Bodden, E. An In-Depth Study of More Than Ten Years of Java Exploitation. In Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS'16), Vienna, Austria, 24–28 October 2016; pp. 779–790.

9. Mastrangelo, L.; Ponzanelli, L.; Mocci, A.; Lanza, M.; Hauswirth, M.; Nystrom, N. Use at Your Own Risk: The Java Unsafe API in the Wild. In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15), Pittsburgh, PA, USA, 25–30 October 2015; pp. 695–710.
10. CVE-2012-0507. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0507 (accessed on 8 October 2017).
11. CVE-2012-5088. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-5088 (accessed on 8 October 2017).
12. CVE-2013-0422. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2013-0422 (accessed on 8 October 2017).
13. CVE-2018-1000146. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000146 (accessed on 8 May 2018).
14. CVE-2017-12617. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-12617 (accessed on 8 March 2018).
15. CVE-2017-12615. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-12615 (accessed on 8 March 2018).
16. CVE-2014-0116. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0116 (accessed on 8 March 2017).
17. CVE-2014-0113. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0113 (accessed on 8 March 2017).
18. CVE-2014-0112. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0112 (accessed on 8 March 2017).
19. CVE-2014-0094. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0094 (accessed on 8 March 2017).
20. CVE-2014-0114. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0114 (accessed on 8 March 2017).
21. CVE-2013-4444. Available online: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4444 (accessed on 8 October 2017).
22. CVE-2016-3088. Available online: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3088 (accessed on 8 October 2017).
23. CVE-2010-1622. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1622 (accessed on 8 March 2017).
24. CVE-2012-5076. Available online: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-5076 (accessed on 8 October 2017).
25. Sailer, R.; Zhang, X.; Jaeger, T.; van Doorn, L. Design and Implementation of a TCG-based Integrity Measurement Architecture. In Proceedings of the 13th Conference on USENIX Security Symposium (SSYM'04), San Diego, CA, USA, 9–13 August 2004; USENIX Association: Berkeley, CA, USA, 2004; Volume 13, pp. 223–238.
26. Jaeger, T.; Sailer, R.; Shankar, U. PRIMA: Policy-reduced Integrity Measurement Architecture. In Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT'06), Lake Tahoe, CA, USA, 7–9 June 2006; pp. 19–28.
27. Xu, W.; Zhang, X.; Hu, H.; Ahn, G.J.; Seifert, J.P. Remote Attestation with Domain-Based Integrity Model and Policy Analysis. *IEEE Trans. Dependable Secure Comput.* **2012**, *9*, 429–442. [CrossRef]
28. Xing, B.; Han, Z.; Chang, X.; Liu, J. OB-IMA: out-of-the-box integrity measurement approach for guest virtual machines. *Concurr. Comput. Pract. Exp.* **2015**, *27*, 1092–1109. [CrossRef]
29. Thober, M.; Pendergrass, J.A.; Jurik, A.D. JMF: Java Measurement Framework: Language-supported Runtime Integrity Measurement. In Proceedings of the 7th ACM Workshop on Scalable Trusted Computing (STC'12), Raleigh, NC, USA, 15 October 2012; pp. 21–32.
30. Zhang, T.; Lee, R.B. CloudMonatt: An Architecture for Security Health Monitoring and Attestation of Virtual Machines in Cloud Computing. In Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15), Portland, OR, USA, 13–17 June 2015; pp. 362–374.

31. Ren, J.; Liu, L.; Zhang, D.; Zhang, Q.; Ba, H. Tenants Attested Trusted Cloud Service. In Proceedings of the 2016 IEEE 9th International Conference on Cloud Computing (CLOUD'16), San Francisco, CA, USA, 27 June–2 July 2016; pp. 600–607.
32. Ba, H.; Zhou, H.; Ren, J.; Wang, Z. Runtime Measurement Architecture for Bytecode Integrity in JVM-Based Cloud. In Proceedings of the 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS'17), Hong Kong, China, 26–29 September 2017; pp. 262–263.
33. TIOBE Index for May 2018. Available online: https://www.tiobe.com/tiobe-index// (accessed on 6 June 2018).
34. Beneke, T.; Wieldt, T. JavaOne 2013 Review: Java Takes on the Internet of Things. Available online: http://www.oracle.com/technetwork/articles/java/afterglow2013-2030343.html (accessed on 25 September 2017).
35. Gosling, J. Java Intermediate Bytecodes. In Proceedings of the 1995 ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, 22 January 1995; pp. 111–118.
36. Lindholm, T.; Yellin, F. *Java Virtual Machine Specification*, 2nd ed.; Addison-Wesley Longman Publishing Co., Inc.: Chicago, IL, USA, 1999.
37. Java ClassFile Format. Available online: http://commons.apache.org/proper/commons-bcel/manual/jvm.html (accessed on 25 September 2017).
38. Smith, S.W. Outbound Authentication for Programmable Secure Coprocessors. In Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS'02), Zurich, Switzerland, 14–16 October 2002; Springer: Berlin, Germnay, 2002; pp. 72–89.
39. Smith, S.W. Outbound authentication for programmable secure coprocessors. *Int. J. Inf. Secur.* **2004**, *3*, 28–41. [CrossRef]
40. Berger, S.; Cáceres, R.; Goldman, K.A.; Perez, R.; Sailer, R.; van Doorn, L. vTPM: Virtualizing the Trusted Platform Module. In Proceedings of the 15th Conference on USENIX Security Symposium (USENIX-SS'06), Vancouver, BC, Canada, 31 July– 4 August 2006; USENIX Association: Berkeley, CA, USA, 2006; pp. 305–320.
41. Liu, Y.; Zhou, T.; Chen, K.; Chen, H.; Xia, Y. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15), Denver, CO, USA, 12–16 October 2015; pp. 1607–1619.
42. Zhang, T.; Lee, R.B. Design, Implementation and Verification of Cloud Architecture for Monitoring a Virtual Machine's Security Health. *IEEE Trans. Comput.* **2018**, *67*, 799–815. [CrossRef]
43. Ba, H.; Wang, Z.; Ren, J.; Zhou, H. JVM-Based Dynamic Attestation in Cloud Computing. In Proceedings of the 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'14), Beijing, China, 24–26 September 2014; pp. 929–934.
44. Ba, H.; Ren, J.; Wang, Z.; Zhou, H.; Li, Y.; Hong, T. User-policy-based dynamic remote attestation in cloud computing. *Int. J. Embed. Syst.* **2016**, *8*, 39–45. [CrossRef]
45. SPECjvm2008. Available online: https://www.spec.org/jvm2008/ (accessed on 10 December 2017).
46. Hou, K.Y.; Shin, K.G.; Sung, J.L. Application-assisted Live Migration of Virtual Machines with Java Applications. In Proceedings of the Tenth European Conference on Computer Systems (EuroSys'15), Bordeaux, France, 21–24 April 2015; pp. 1–15.
47. Bull, J.M.; Smith, L.A.; Westhead, M.D.; Henty, D.S.; Davey, R.A. A Methodology for Benchmarking Java Grande Applications. In Proceedings of the ACM 1999 Conference on Java Grande (JAVA'99), San Francisco, CA, USA, 12–14 June 1999; pp. 81–88.
48. Gil, J.Y.; Lenz, K.; Shimron, Y. 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 (SPLASH'11 Workshops), Portland, OR, USA, 23–24 October 2011; pp. 297–308.
49. Blackburn, S.M.; Garner, R.; Hoffmann, C.; Khang, A.M.; McKinley, K.S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S.Z.; et al. 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 (OOPSLA'06), Portland, OR, USA, 22–26 October 2006; pp. 169–190.
50. Garfinkel, T.; Rosenblum, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In Proceedings of the Network and Distributed System Security Symposium (NDSS'03), San Diego, CA, USA, 6–7 February 2003; Internet Society: Reston, VA, USA, 2003; pp. 191–206.

51. Haldar, V.; Chandra, D.; Franz, M. Semantic Remote Attestation: A Virtual Machine Directed Approach to Trusted Computing. In Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium (VM'04), San Jose, CA, USA, 6–7 May 2004; USENIX Association: Berkeley, CA, USA, 2004; pp. 3–18.
52. Mei, S.; Wang, Z.; Cheng, Y.; Ren, J.; Wu, J.; Zhou, J. Trusted Bytecode Virtual Machine Module: A Novel Method for Dynamic Remote Attestation in Cloud Computing. *Int. J. Comput. Intell. Syst.* **2012**, *5*, 924–932. [CrossRef]
53. Shankar, U.; Jaeger, T.; Sailer, R. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In Proceedings of the Network and Distributed System Security Symposium (NDSS'06), San Diego, CA, USA, 2–3 February 2006; Internet Society: Reston, VA, USA, 2006.
54. Shi, J.; Yang, Y.; Tang, C. Hardware Assisted Hypervisor Introspection. *SpringerPlus* **2016**, *5*, 647–669. [CrossRef] [PubMed]
55. Pistoia, M.; Banerjee, A.; Naumann, D.A. Beyond Stack Inspection: A Unified Access-Control and Information-Flow Security Model. In Proceedings of the 28th IEEE Symposium on Security and Privacy (SP'07), Berkeley, CA, USA, 20–23 May 2007; pp. 149–163.
56. Holzinger, P.; Hermann, B.; Lerch, J.; Bodden, E.; Mezini, M. Hardening Java's Access Control by Abolishing Implicit Privilege Elevation. In Proceedings of the 38th IEEE Symposium on Security and Privacy (SP'17), San Jose, CA, USA, 22–26 May 2017; pp. 1027–1040.
57. Sridharan, M.; Artzi, S.; Pistoia, M.; Guarnieri, S.; Tripp, O.; Berg, R. F4F: Taint Analysis of Framework-based Web Applications. In Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'11), Portland, OR, USA, 22–27 October 2011; pp. 1053–1068.
58. Bell, J.; Kaiser, G. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14), Portland, OR, USA, 20–24 October 2014; pp. 83–101.
59. Focardi, R.; Palmarini, F.; Squarcina, M.; Steel, G.; Tempesta, M. Mind Your Keys? A Security Evaluation of Java Keystores. In Proceedings of the Network and Distributed System Security Symposium (NDSS'18), San Diego, CA USA, 18–21 February 2018; Internet Society: Reston, VA, USA, 2018; pp. 1–15.