MUSTI: Dynamic Prevention of Invalid Object Initialization Attacks

Alexandre Bartel, Jacques Klein, Yves Le Traon
University of Luxembourg, SnT
Luxembourg, Luxembourg

***Abstract*—Invalid object initialization vulnerabilities have been identified since the 1990's by a research group at Princeton University. These vulnerabilities are critical since they can be used to totally compromise the security of a Java virtual machine. Recently, such a vulnerability identified as CVE-2017-3289 has been found again in the bytecode verifier of the JVM and affects more than 40 versions of the JVM. In this paper, we present a runtime solution called MUSTI to detect and prevent attacks leveraging this kind of critical vulnerabilities. We optimize MUSTI to have a runtime overhead below 0.5% and a memory overhead below 0.42%. Compared to state-of-the-art, MUSTI is completely automated and does not require to manually annotate the code.**

***Index Terms*—Java, object initialization, vulnerability, sandbox**

I. INTRODUCTION

WHEN the Java language was introduced in the mid-1990's, it was thought that the language is more secure than C/C++ because it does not allow to directly manipulate the memory – it uses a garbage collector instead – and because array bounds are automatically checked at runtime. This design makes certain vulnerabilities such as buffer overflows a thing of the past. Unfortunately, the Java Virtual Machine (JVM) and part of the Java library are still written in C/C++ code which makes the whole Java architecture still vulnerable to low level attacks.

Another Java feature, emphasized by Sun Microsystems at the time, is the fact that the JVM can run untrusted code in a sandbox and give this untrusted code only limited or no privilege at all. This was particularly convenient in web browser which could execute such untrusted code in so called *Applet*s [5] with the least privileges. Alas, even though the security architecture seemed fine, numerous security vulnerabilities have been found in Java which enable, in most cases, a total sandbox escape. This means that if an analyst [1] can redirect a user to a web page he controls, he can run malicious Java code on the user's browser to escape the sandbox and potentially run code with the privileges of the web browser. The situation was so alarming for Java and other plugins based on the NPAPI, that major companies developing web browsers such as Google [12] and Mozilla [13] decided to first disable them by default and to then remove them altogether.

Today, a typical computer user navigates the world wide web without executing any Java code within his browser. However, it may still be the case that companies who rely on legacy software require their employees to activate the Java plugin in their browsers to access specific services. Also, some computer users – not necessarily within a company – may also choose to re-enable Java to access a particular service. Forcing the browser to use Java increases the attack surface and thus puts users at risk.

Not only are end users at risk of having their machines compromised but so are servers. Indeed, some services such as Apache Spark [2] (an analytics engine for big data) run user provided Java code. Even if the code is being run in a sandbox, it might end up running malicious code to exploit a security vulnerability to escape the Java sandbox. In this paper we focus on one kind of such critical vulnerabilities called *invalid object initialization*. Our approach aims at improving the Java virtual machine to prevent invalid object initialization vulnerabilities from being exploited at runtime.

Meanwhile, Oracle [3] did a lot of effort to improve the security of the Java platform. One approach they used is to reduce the attack surface. Indeed, the Java Class Library (JCL) has numerous legacy classes which are prone to contain security vulnerabilities. By marking them as "restricted" an analyst cannot directly instantiate them anymore and thus cannot use code that might have been useful to perform his attack. This effectively breaks existing exploits relying on such classes but also makes it harder for the analyst to find new code he can leverage in a new attack.

Oracle also developed an approach to automate the process of verifying the code and finding new security vulnerabilities [20]. The approach taints untrusted user data and checks if it flows to security sensitive operations such as the loading of a class in a privileged context. If it does, the approach makes sure that objects created by the security sensitive operation do not flow back to the user context. This prevents, in our example, an analyst from using a class loaded by a security sensitive operation.

Unfortunately, despite these efforts, new vulnerabilities have been found. One of the latest vulnerabilities, CVE-2017-3289 – an invalid object initialization vulnerability – is studied in this paper. This kind of vulnerability is critical as it allows an analyst to completely bypass the Java sandbox.

Researchers have already developed an approach to tackle this kind of vulnerability [29]. However, it requires to man-

[1] Instead of using the term *attacker*, in this paper, we choose the more neutral term *analyst* which represents both an attacker and a researcher.

[2] http://spark.apache.org/

[3] Oracle completed the acquisition of Sun Microsystems in 2010.

ually annotate the code which makes it difficult to use when the code is in constant evolution as is the case for the JCL and the JVM. This might explain why such an approach has not yet been adopted in practice. On the contrary, our approach is fully automated and does not require any human intervention.

The contributions we make are the following:

- We analyze a recent Java vulnerability and develop proof-of-concept for it
- We analyze the root causes leading to invalid object initialization vulnerabilities
- We propose an open-source and fully automated solution, MUSTI, to prevent Java sandbox bypasses leveraging uninitialized instance vulnerabilities [4]
- We evaluate MUSTI in terms of detection, runtime overhead and memory overhead

This paper is organized as follows. First, in Section II, the Java security model is presented. Then, we explain what the invalid object initialization vulnerability is in Section III. In Section IV we present our approach to prevent the vulnerability at runtime. We evaluate our approach in Section V. In Sections VI and VII we describe the limitations of the approach and discuss potential improvements. Related work is presented in Section VIII. Finally, we conclude in Section IX.

II. THE JAVA SECURITY MODEL

In this section, we briefly present the fundamental concepts that are required to understand the Java security model: security policy, security domains, permissions and the security manager. The reader familiar with the Java security model can directly jump to Section III.

A. Security Policy

Java code can be associated with a security policy. The policy is a list of permissions describing what the code is allowed to do. For instance, a security policy containing the `SOCKET` permission allows the associated Java code to connect to a remote host; a security policy containing the `CREATE_CLASS_LOADER` permission allows the associated Java code to create a class loader. Usually, it makes sense to give more permissions to trusted code and as few permissions as possible to untrusted code. For instance, trusted server code can run with all permissions, while untrusted code running in the browser runs with no permission. To prevent the code from performing forbidden operations, Java runs untrusted code within a sandbox. For every sensitive operation the code tries to perform, the sandbox checks at runtime that the code is authorized. If it is not, a security exception is thrown. Untrusted code typically has no permission and, thus, cannot access the file system, the network, etc.

B. Security Domain

Every class in the JVM is loaded with a class loader and associated with a security domain. Classes shipped with the JRE (Java Runtime Environment) also known as *system* classes, are loaded with all permissions. An example of a system class is `java.lang.Class`. Untrusted classes downloaded from the Internet and running in an applet, or more generally all classes coming from an untrusted source, should be loaded with no permission. Trusted classes can be loaded with all permissions but should be loaded with the least permissions to respect the principle of least privilege [33].

In this paper we suppose that the analyst is running untrusted code on the Java virtual machine. This untrusted code runs within the sandbox and has no permission.

System Classes (Trusted code)

```
class ClassLoader {                                                    1
  protected ClassLoader() {                                            2
    this(checkCreateClassLoader(), getSystemClassLoader())             3
        ;}
  }                                                                    4
                                                                       5
  private static Void checkCreateClassLoader() {                       6
    SecurityManager security = System.getSecurityManager();           7
    if (security != null) {                                            8
      security.checkCreateClassLoader();                               9
    }                                                                  10
    return null;                                                       11
  }                                                                    12
}                                                                      13
                                                                       14
class SecurityManager {                                                15
  public void checkCreateClassLoader() {                               16
    checkPermission(SecurityConstants.                                 17
        CREATE_CLASSLOADER_PERMISSION);
  }                                                                    18
}                                                                      19
```

Application Classes (Untrusted code)

```
class UntrustedMain {                                                  20
  public static void main(String[] args) {                            21
    ClassLoader myCL = new ClassLoader() { };                          22
  }                                                                    23
}                                                                      24
```

Fig. 1. Code without the `CREATE_CLASSLOADER` permission (UntrustedMain) cannot instantiate a class loader (line 22) because the security check in the system class `java.lang.ClassLoader` (line 17) will throw a security exception.

C. The Security Manager

Permissions are only checked when a security manager has been created and set. This can be done programmatically via a call to `System.setSecurityManager()` or with a command line option when launching the Java virtual machine. How the security manager is used when checking permissions is illustrated in Figure 1. In the constructor of the system class `ClassLoader` from the `java.lang` package, there is a call to `checkCreateClassLoader()` (line 3). This method then calls `checkCreateClassLoader` of the security manager (line 9). Finally, the security manager calls `checkPermission()` to check for permission `CREATE_CLASSLOADER` (line 17). Notice that the security check is only performed if a security manager is set (lines 7-8). Thus, with a security manager set, untrusted code cannot instantiate a subclass of a `ClassLoader` since the constructor checks for the `CREATE_CLASSLOADER` permission.

[4]The implementation is available at https://github.com/Alexandre-Bartel/jvm-musti.

SecurityManager.checkPermission()
SecurityManager.checkCreateClassLoader() (line 16)
ClassLoader.checkCreateClassLoader() (line 6)
ClassLoader() (line 2)
UntrustedMain.main() (line 21)

Fig. 2. Call stack when the `checkPermission` method is called (Figure 1 line 17).

D. Permission Checks

When a permission P is checked, all elements of the stack trace (all methods that have been called since `main()`) are analyzed and must hold permission P. Otherwise, a `SecurityException` is thrown.

When the code of Figure 1 is executed and reaches line 17, it has the call stack with five elements illustrated Figure 2. The `checkPermission` method goes backwards when analyzing the call stack. The three last methods `ClassLoader()`, `checkCreateClassLoader()` and `checkCreateClassLoader()` are from system classes and thus, have all permissions. Method `main()`, however, is from of an untrusted class, and does not have the `CREATE_CLASSLOADER` permission. Thus, `checkPermission` throws a `SecurityException`, which prevents the `ClassLoader` from being instantiated.

III. Uninitialized Instance Vulnerability

A. What it is

An invalid object initialization vulnerability enables the creation of an object which is not properly initialized. In the case of Java, this means that the chain of calls to constructors is broken resulting in some constructor methods not being called. The consequences are the following:

- code that should be executed may not be executed
- fields that should be initialized may not be initialized and may thus end up having "default" values (e.g., *null* for references)

Take the example of Figure 3 representing the class hierarchy of hypothetical Java library classes A, B and C. In this example, class C extends A and classes A and B both extend $Object$. In a normal program instantiating a new object of type C, the constructor of C starts executing. The first instruction of the constructor actually calls the constructor of A (Figure 4 line 11 right) which immediately calls the constructor of $Object$ (Figure 4 line 10 left). When the constructor of $Object$ terminates, the execution goes back in the constructor of A which continues and terminates. Finally, the constructor C continues and also terminates.

An invalid object initialization vulnerability allows to create an instance of an object whose constructor will not call the constructor of its super class (e.g., by exploiting a bug in the bytecode verifier). More concretely, if we have class E extending A and class D extending B (Figure 3), the vulnerability allows to create objects of type E without calling A's constructor or objects of type D without calling B's constructor.

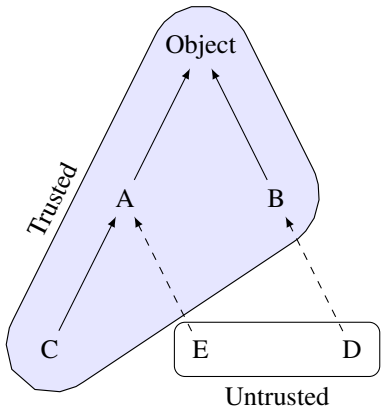

Fig. 3. Class hierarchy for Java library classes A, B and C (representing trusted code) and application defined classes D and E (representing untrusted code).

```
1  class Object {
2   <init>() {
3     ...
4   }
5  }
6
7  class A {
8   bool forbidden;
9   <init>() {
10    super();
11    forbidden = true;
12    ...
13  }
14  void m() {
15    if (forbidden) {
16      return;
17    }
18    ...
19  }
```

```
1  class B {
2   <init>() {
3     super();
4     checkPermission("P1");
5     ...
6   }
7  }
8
9  class C {
10  <init>() {
11    super();
12    ...
13  }
14 }
```

Fig. 4. Constructor code for classes $Object$, A, B and C. Note that constructor B is checking for permission $P1$ and method m from class A is using a field initialized in a constructor for access control.

With an invalid object initialization vulnerability an analyst can create a subclass whose constructor does not call the super class constructor.

B. Impact on Security

If access control or other security mechanisms rely on the value of fields initialized in constructors, an analyst could be able to bypass them by setting the field values to default values. Likewise, if security checks are directly performed in constructor code an analyst may be able to bypass them by not executing the code.

Figure 4 illustrates these two kinds of vulnerable code. The authorization check in the constructor is illustrated in the constructor of class B. If class B has a subclass such as D, controlled by the analyst, the constructor of B will not be called from D during an attack. Thus, the permission checked within the code of the constructor of B will not be called. The consequence is that the analyst can instantiate objects he should not be able to because he can bypass permission checks in the code of their constructors. Without the invalid object

initialization vulnerability, the instantiation of such object would have thrown a security exception at runtime.

In the original Princeton attack [22], the invalid object initialization vulnerability was used to call a normally unreachable method to define an analyst controlled class with all privileges. This class then disables the security manager to bypass the Java sandbox. The normally unreachable method in question is `defineClass` from the `ClassLoader` class. This method is *protected* and can thus only be accessed from subclasses. Since the constructors of `ClassLoader` are checking that the caller has the appropriate permission, an analyst (running code in a sandbox without permission) should not be able to create a subclass. However, with the invalid object initialization vulnerability, the analyst bypasses the permission check when he creates a subclass and can thus have access to the `defineClass` method to bypass the sandbox.

For the sake of completeness, we also describe here the impact of uninitialized fields. The field used as condition for access control is illustrated in method $A.m$ (lines 14-18, left). If class A has a subclass such as E, controlled by the analyst, the constructor of E will not call the constructor of A. The consequence is that field $forbidden$ will have the default value of $false$. Any subsequent call to method $A.m$ will succeed since the access control check is bypassed. Note that we assume it is bad practice and quite rare to use fields for access control within the Java sandbox. Finding such fields is thus out of the scope of this paper.

> Not calling the super class constructor allows an analyst to create a subclass S which bypasses security checks done in S's superclass constructor's code and hence allows the analyst to have access to privileged methods of the superclass.

C. Vulnerability History

As far as we know, there are at least three publicly known *invalid object initialization* vulnerabilities for Java. The first publicly known invalid object initialization vulnerability has been found by a research group at Princeton in 1996 [22]. The vulnerability lies within the bytecode verifier. It allows for a constructor to catch exceptions thrown by a call to `super()` and return a partially initialized object. Note that at the time of this attack the class loader class did not have any instance variable. Thus, leveraging the vulnerability to instantiate a class loader gave a fully initialized class loader on which any method could be called. The second has been discovered by LSD [5] in 2002 [26]. The authors also exploited a vulnerability in the bytecode verifier which enables to not call the constructor of the super class. They have not been able to develop an exploit to completely escape the sandbox. They were able, however, to access the network and read and write files to the disk. The last one has been made public in 2017 and is CVE-2017-3289. This vulnerability is also a bug in the bytecode verifier and its description indicates that an analyst can completely bypass the Java sandbox.

[5] a security research group called The Last Stage of Delirium

It is not surprising that invalid object initialization vulnerabilities are all located within the bytecode verifier. Indeed, it is the bytecode verifier which is in charge of validating that constructors do not return uninitialized objects. The code of the bytecode verifier evolves constantly through refactoring or the implementation of new functionalities. Unfortunately, every modification of this code could introduce a bug whose consequence is that bytecode properties are not properly enforced. Hence, by exploiting such a bug, it might be possible to find a path on which a constructor might return an uninitialized object. MUSTI detects such uninitialized objects at runtime to prevent critical attacks which could compromise the security of the JVM.

D. Concrete Example

In this section, we concretely describe how an analyst could bypass permission checks by exploiting an invalid object initialization vulnerability. The real-world concrete context is the following: the analyst can submit jobs (represented by Java classes with no permission) to a target Java application (for instance a Java Applet or the Apache Spark application) running on a Java VM vulnerable to an invalid object initialization vulnerability. We use CVE-2017-3289 as the target vulnerability as this is a recent invalid object initialization vulnerability. However, the approach is similar with other invalid object initialization vulnerabilities.

The description of CVE-2017-3289 indicates that *"Successful attacks of this vulnerability can result in takeover of Java SE, Java SE Embedded."* [8]. This means it is possible to exploit the vulnerability to escape the Java sandbox (and thus bypass all permission checks). Redhat's bugzilla indicates that *"An insecure class construction flaw, related to the incorrect handling of exception stack frames, was found in the Hotspot component of OpenJDK. An untrusted Java application or applet could use this flaw to bypass Java sandbox restrictions."* [2]. This informs the analyst that (1) the vulnerability lies in C/C++ code (Hotspot is the name of the Java VM) and that (2) the vulnerability is related to an illegal class construction and to exception stack frames. Information (2) indicates that the vulnerability is probably in the C/C++ code checking the validity of the bytecode. The page also links at the OpenJDK's patch for this vulnerability.

We reverse engineered the patch (see our technical report for more details [15]) and were able to create a working exploit code illustrated in Figure 5. An analyst can use this code to subclass, for instance, a system class which checks for permission P in its constructor. When the subclass is instantiated, the constructor of the superclass (in our case, the system class constructor checking for permission P) is not executed. Thus, the analyst now has an instance of the system class and can call method on this instance even though this should not have been possible because the analyst's code has no permission.

As presented in Section II, Java permission protect system elements such as the filesystem, network or code execution. By exploiting invalid object initialization vulnerabilities, an analyst could – without any permission – access the filesystem, network or execute arbitrary code. More concretely, by

```
1  <init>()
2    try_start:
3      new "java/lang/Throwable"
4      dup
5      invokespecial "Throwable.<init>()"
6      athrow
7      // locals[0] = UNINITIALIZED_THIS
8      // stack[0] = "java/lang/Throwable"
9    try_end:
10   handler:
11     // locals[0] = TOP
12     // stack[0] = "java/lang/Throwable"
13     return
```

Fig. 5. Proof-of-Concept for exploiting the uninitialized instance vulnerability in the bytecode verifier. In this example the constructor returns an instance that is still flagged with `UNINITIALIZED_THIS`.

exploiting the vulnerability, an analyst is able to, for instance, instantiate his own classloader by bypassing the permission check (see the `java.lang.ClassLoader` constructor in Figure 1 as well as the concrete code here: https://github.com/Alexandre-Bartel/attack-musti) and would thus be able to define his own classes with all permissions. This is the reason why the vulnerability has been classified as critical by Oracle.

E. A Threat for which Software Environments?

Invalid object initialization vulnerabilities might only affect object-oriented languages such as Java or C++. Furthermore, invalid object initialization vulnerabilities we target in this paper can only be found in systems such as the JVM which provide a sandbox running in the same process and written in the same language as the "untrusted" code. We are aware of a similar sandboxing system at least for Microsoft's C#.

In contrast, in the "Android" world where applications are also running on a Java-like VM, authorization checks are not done on the same VM as the "untrusted" code (applications) but in another process [18]. Thus, an invalid object initialization vulnerability could not impact the authorization checks (unless the checks rely on objects that the untrusted code can create, but this would not make much sense). In a nutshell, and as far as we know, this kind of vulnerability currently mainly affects all implementations of the Java virtual machine such as Oracle's OpenJDK, IBM's J9 [4] or Excelsior JET [3] as well as all applications running on top of the VM allowing "untrusted code" to run in the sandbox. In theory, it could also affect the C# virtual machine [32], but no such vulnerability has yet been found. The process of initializing objects in the C# bytecode being simpler, the code might contain less bugs and thus less vulnerabilities. Last, but not least, it could also affect all applications running on top of these VMs allowing "untrusted code" to run in the sandbox.

IV. Preventing the Vulnerability

Our generic approach aims at preventing the exploitation of invalid object initialization vulnerabilities at runtime. We patch the Java virtual machine to add code which checks that objects have been correctly initialized, i.e. that the chain of constructors has not been broken. To understand where to patch the virtual machine we first have to understand how it loads and represents the code. Note that as our solution is generic, it could be implemented for another object-oriented language with a similar sandbox system as the Java virtual machine, for example C# [32].

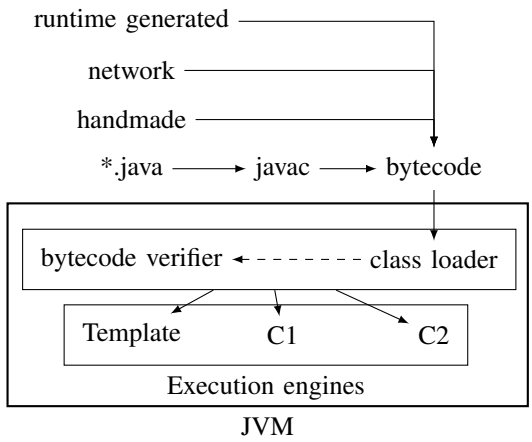

Fig. 6. Simplified View of the Java Runtime: The JVM loads bytecode and may verify it before it is executed by one of the execution engines ($Template$, $C1$ or $C2$)

A. Code Loading in the JVM

As illustrated in Figure 6, the Java Virtual Machine (JVM) loads only bytecode. The bytecode however, can originate from multiple sources: compiled by the Java compiler (the usual), downloaded from the network, generated at runtime, or assembled manually (typical for exploiting a vulnerability in the bytecode verifier). The JVM loads the bytecode through a class loader. The bytecode is usually verified but not for classes which are deemed "trusted" such as classes in the packages "java.*". At runtime, the bytecode of a Java method is either executed with the template engine, the C1 engine or the C2 engine. The C1 and C2 engines transform and optimize the bytecode and execute the resulting code. Which engine is chosen depends on the number of times the method has already been executed. The appropriate place to instrument the bytecode is thus within the class loader. Indeed, this is where the virtual machine loads all class and creates an internal representation of the class' bytecode. Since every bytecode that is loaded by the JVM has to go through the class loader, we instrument the bytecode there.

B. Instrumenting Code in the JVM

Naively instrumenting existing bytecode may result in broken bytecode. Indeed, the instrumented bytecode must verify:

- that branching instruction offsets are still pointing to the right instruction,
- that try/catch blocks and handlers are still consistent
- that stack map frames [6] are appropriate and still consistent

Our approach is to add one field, $is\_initialized$ to the $Object$ class and to modify the bytecode of the constructors

[6] structures to help type checking the bytecode [10]

```
1  aload_0
2  ...
3  return
4  ...
5  return
```

```
1  aload_0
2  aconst_1
3  putfield allow_in_constructor = 1
4  aload_0
5  ...
6  goto new_label
7  ...
8  goto new_label
9  new_label:
10 aload_0
11 aconst_1
12 putfield is_initialized = 1
13 aload_0
14 aconst_0
15 putfield allow_in_constructor = 0
```

Fig. 7. Constructor Transformation (pseudo bytecode). The original constructor code is shown on the left. Transformed or additional code has a gray background color.

```
1  aload_0
2  ...
3  return
4  ...
5  return
```

```
1  aload_0
2  invokevirtual isInit()
3  ifne new_label
4  new SecurityException
5  athrow
6  new_label:
7  aload_0
8  ...
9  return
10 ...
11 return
```

Fig. 8. Method Transformation (pseudo bytecode). The original method code is shown on the left. Transformed or additional code has a gray background color.

of the $Object$ class to initialize the field to $true$. When a method is called, it could thus first check that the field is correctly initialized. If it is, the method is executed normally. Otherwise, it means that the object on which the method is called has not been correctly initialized. The method does not execute and the program stops, for instance by throwing an exception.

a) Object Constructors Instrumentation: Only constructors in `java.lang.Object` are instrumented. The modifications are presented in Figure 7. First, code is prepended to the constructor bytecode to set the field $allow\_in\_constructor$ to $true$ (lines 1 to 3). This allows the constructor to call methods even if it is not fully initialized yet. Then, every `return` instruction is changed to a `goto` instruction to branch to the appended code. The appended code (lines 9 to 15) sets back the field $allow\_in\_constructor$ to $false$ and sets the field $is\_initialized$ to $true$ indicating that the object has been correctly initialized.

b) Method Instrumentation: The bytecode transformation for non-constructor methods is illustrated in Figure 8. Some code is prepended to the method bytecode to check if the constructor has been properly initialized (lines 1 and 2). The check is done via a call to method `isInit()` which returns $true$ if the object has been properly initialized. If it is the case, the method is executed normally (lines 3 and 6). Otherwise, the method throws a security exception (lines 4 and 5).

C. Implementation of MUSTI

We use the Java virtual machine from OpenJDK 8 update 144 branch 01 (see Appendix A for more details).

We modify `method.cpp` to add code which modifies the bytecode of existing methods. We rely on code already present in the JVM file `relocator.cpp` to instrument the bytecode of methods. In theory, we modify `classFileParser.cpp` to add code to instrument the constructors of the `java.lang.Object` class. In practice, as explained in the next paragraph, we modify `classFileParser.cpp` to instrument all constructors of all classes directly extending `java.lang.Object`. Overall, the new code accounts for about 2000 lines of C++.

While implementing our solution we faced one major challenge which is that the `java.lang.Object` class cannot be easily modified. According to multiple sources [7] and our experience, adding a field or a method to this class would require heavy modification of the Java virtual machine source code since numerous parameters for this class are hardcoded throughout the source code of the virtual machine. We solved this by modifying all classes which immediately extend `java.lang.Object`.

V. EVALUATION

In this section, we answer the following research questions:

- RQ1: can MUSTI prevent attacks based on unitialized instance vulnerabilities?
- RQ2: what is the runtime overhead of MUSTI?
- RQ3: what is the memory overhead?
- RQ4: how many constructors are vulnerable?

All the experiments were performed on a machine with 32Gb of RAM and an Intel Core i7-6700HQ CPU @ 2.60GHz featuring 8 processors each having 8 cores.

A. RQ1: Preventing Attacks

The main goal of MUSTI is to prevent attacks based on invalid object initialization. We reverse engineered the patch of vulnerability CVE-2017-3289 to create an exploit. This exploit features unprivileged code which leverages the vulnerability to create an instance of `java.lang.ClassLoader`. How the vulnerability has been reversed and the exploit created is detailed in the technical repport [15].

The exploit code has no permission and can nonetheless create an instance of `java.lang.ClassLoader` on a vulnerable version of the Java virtual machine. However, in our modified version MUSTI in which we injected the vulnerable code in the bytecode verifier, the added bytecode in the `java.lang.ClassLoader` constructor detects that the constructor call chain has been broken since the field $is\_initialized$ is still set to $false$. It thus successfully stops the program before it can leverage the vulnerability.

> MUSTI is able to successfully prevent attacks leveraging invalid object initialization vulnerabilities present within the bytecode verifier.

[7] https://stackoverflow.com/add-a-field-to-java-lang-object
https://stackoverflow.com/instrumenting-array-via-java-lang-object

B. RQ2: Runtime Overhead

In Section V-B1 and Section V-B2, we first evaluate MUSTI on DaCapo [19], a benchmark suite intended as a tool for Java benchmarking, as well as on two real world Java programs: Soot [30] and JavaML [14]. For every target test suite or program, we run the program 50 times in a row in the same Java virtual machine. We do this to be able to stabilize the running time of the Java virtual machine. Indeed, the JVM is quite a complex software which requires to load classes used in the target program and which comes with many optimizations to improve the running time. In order to evaluate a target program in its best optimized version and to remove noise related to class loading and code optimization, we run it 50 times and use the last 10 runs as representative of the best optimized version of the program. Every run with 50 iterations is repeated 10 times. The final version uses the mean running time of every iteration.

We then evaluate the class loading overhead of MUSTI based on the DaCapo benchmark in Section V-B3.

1) DaCapo Benchmark: The DaCapo benchmark is not maintained anymore. Therefore, some of the benchmark's test suites are not working against Java 8. Thus, we only rely on a subset of all the available test suites, namely `luindex`, `lusearch`, `pmd` and `xalan`. The first two are based on Apache Lucene [8]: `luindex` uses lucene to indexes a set of text documents such as the works of Shakespeare, while `lusearch` uses lucene to do a text search of keywords over text documents. Regarding the two other test suites, `pmd` analyzes a set of Java classes to detect potential problems and `xalan` transforms Xml documents into Html documents.

We run all the test suites on both the original JVM, $orig$, and the modified JVM, MUSTI. We use three different versions of the modified JVM. In the first version, $naive$, all methods of all classes are instrumented. In the second version, $opti_1$, only methods of classes which are public and non-final are instrumented [9]. In the third version, $opti_2$, only methods of classes checking for a permission in one of their constructors are instrumented. How the list of such constructors has been computed is the topic of RQ4 in Section V-D.

The results are shown in Figure 9 to 12. Figure 9 represents the results of the experiments for `luindex`. The first graph compares the original JVM, $orig$, (crosses) with the naive modified JVM, $naive$ (circles). The second graph compares $orig$ with $opti_1$ while the third one compares $orig$ with $opti_2$. The overhead is indicated in the three graphs with triangles. Figure 10 represents the result for `lusearch`, Figure 11 for `pmd` and Figure 12 for `xalan`.

The first observation is that the overhead (taken on the last 10 runs) decreases from $naive$ and $opti_1$ to $opti_2$. For `xalan`, for instance, it goes from 9.3% and 10.2% to 2.04%. For `lusearch` it goes from 3.93% and 4.36% to -0.18%. The last overhead is negative meaning that the modified JVM ran faster than the original one. The explanation is that since the run times of $orig$ and $modif$ are very close, it may happen

[8]http://lucene.apache.org

[9]An analyst can only create a subclass from a public and non-final class to perform an invalid object initialization vulnerability attack.

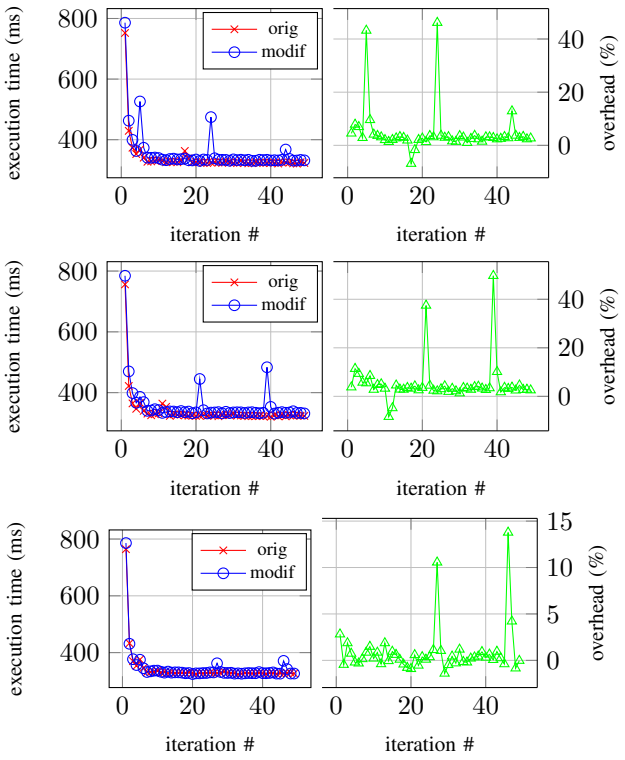

Fig. 9. `luindex` naive (up), opti1 (middle) and opti2 (down)

that the average of $modif$ is faster than the average of $orig$, hence the negative value close to 0%.

The second observation, is that for the first few runs, $modif$ has a very high overhead compared to $orig$. This is explained by the fact that in the modified JVM, the bytecode of a huge number of methods is modified which takes time. This is one of the reason, we execute 50 runs of the same program in the same virtual machine to get rid of the class loading impact on the overall running time of the program. This phenomenon is specially interesting in the case of `xalan`. After iteration 18, the overhead goes from more than 150% down to the range 0-5%. This is not only explained by the method bytecode modification but also by the numerous optimizations the virtual machine performs on the bytecode. These modifications are highly dependent on the number of execution of the method in question, i.e. the more a method is executed, the more the JVM tries to optimize its code. It is likely that for the case of `xalan` a huge batch of method is used frequently and ends up being highly optimized at the 18^{th} iteration.

The third observation is that the overhead of $opti_2$ is close to zero. This means that the modification of the JVM to detect invalid object initialization has almost no impact on the runtime of the program.

The fourth observation is that there are *bursts* of the overhead. This is especially noticeable in Figure 9. We assume this is caused by the garbage collector which takes time to remove all the unused objects and thus increases the running time for some iterations. The negative effect on the overhead can be observed both for the original VM and for MUSTI. Figure 14 illustrates this behavior. At iteration 17, the garbage

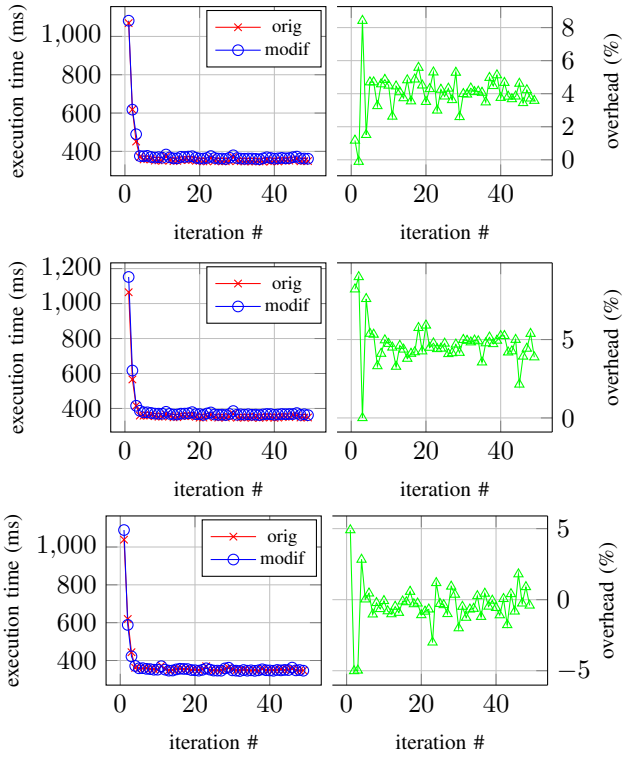

Fig. 10. `lusearch` naive (up), opti1 (middle) and opti2 (down)

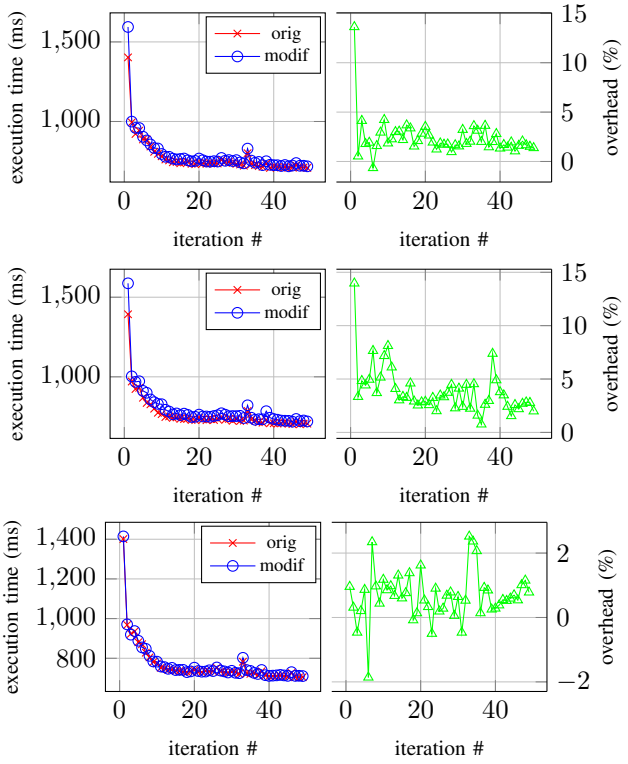

Fig. 11. `pmd` naive (up), opti1 (middle) and opti2 (down)

collector runs for the original VM causing the overhead to be negative. At iteration 19, the garbage collector runs for MUSTI, causing the overhead to be way higher.

> The evaluation on the DaCapo benchmark indicates that impact of MUSTI on the running time is low and is in the range 0-3%.

2) Real Java Software: We also evaluate MUSTI on two real Java programs: Soot [30] a program to analyze and optimize Java bytecode and Java-ML [14] a machine learning library. As for the DaCapo benchmark, we run each program 50 times in the same virtual machine to remove noise from class loading and code optimization. We run Soot with its Dexpler module [17] to transform the 7Mib Dalvik bytecode of one Android application to the internal representation of Soot called Jimple and to output this representation to the file-system. We run Java-ML on the tutorial examples available in the source code: random forest, kmeans, store data, Weka classifier, ARFF loader, Weka clusterer, sampling, feature scoring, feature ranking, feature subset selection, ensemble feature selection, sparse instance, dataset, dense instance, lib SVM, self optimizing lib SVM, KNN, naive Bayes, cross validation, k-dependent Bayes and entropy partitioning. We only run Soot and Java-ML with $opti_2$, the version of the modified JVM which yielded the best results (the lower overheads) for DaCapo. The precise versions of the tools and the Android application are listed in Appendix A.

Figure 13 represents the results for Soot and Figure 14 for Java-ML. The overhead average for the last ten runs is -0.9% for Soot and 0.35 for JavaML.

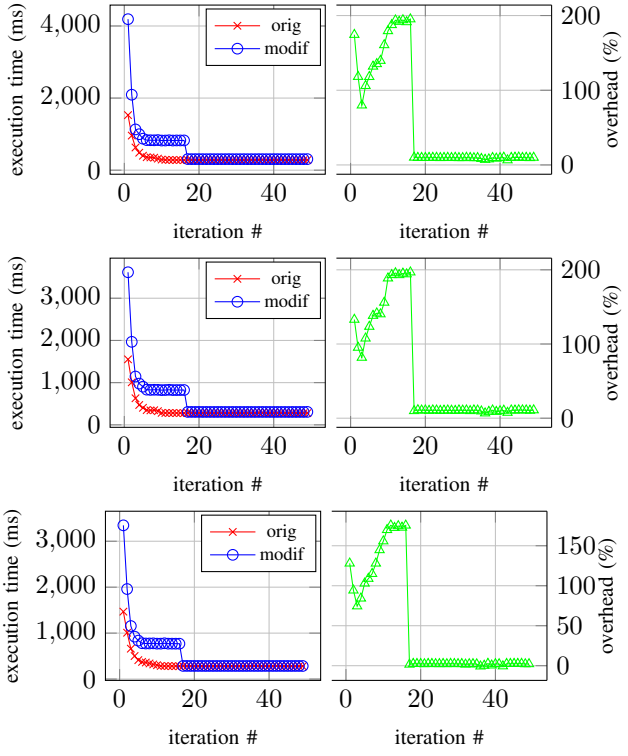

Fig. 12. `xalan` naive (up), opti1 (middle) and opti2 (down)

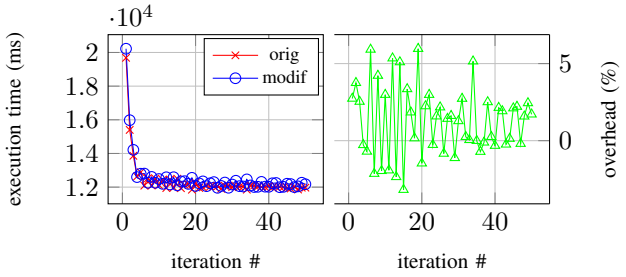

Fig. 13. `Soot` *opti*2

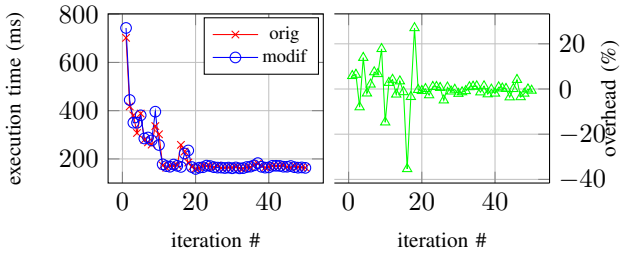

Fig. 14. `JavaML` *opti*2

The impact of MUSTI on the running time of real world Java program is low and less than 0.5%.

3) Class Loading Overhead: To evaluate the runtime overhead of MUSTI's class loading component (in which we instrument classes), we measure the CPU time of the JVM method `ClassFileParser.parseClassFile` responsible for loading, initializing and verifying a Java class. The cumulative loading time of classes from the DaCapo benchmark (about 2500 classes) is represented in Figure 15 on the left. The runtime overhead of class loading is represented in Figure 15 on the right. The overhead is about 32%. This means that class loading CPU time increases from 763 μs in the original JVM to 1011 μs in the modified JVM.

While the overhead appears to be huge, in practice it is insignificant for three main reasons. First, we did not take into account the time to read class files from the disk which is in the order of at least several milliseconds. Second, we measure the overhead for the naive approach where all classes are instrumented. In practice, only about 1.33% of the classes need to be instrumented (see Section V-D). Third, real software such at Soot runs for several seconds (12 seconds in the experiment above with Soot). In that case, the class loading overhead represents less than $\frac{1}{400}$%.

C. RQ3: Memory Overhead

To evaluate the memory overhead we analyze the classes shipped with Java 8 update 144 branch 1. These classes are the basic Java runtime classes such as `java.lang.String` which are present on any Java virtual machine and other classes such as the ones in packages `sun.*`, `com.sun.*` or `javax.*`. The total number of classes is 26,610. They represent approximately 160Mib.

We developed a program based on Soot to count the total number of instructions in all methods of all classes. For every constructor in a non-final non-private class, we add 3 instructions for every return instructions as well as 7 instructions representing the code we append to the constructors. For every non-final non-private method, we add 8 instructions representing the code we prepend to the methods.

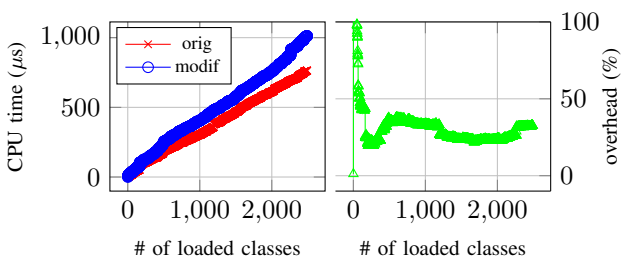

Fig. 15. `ClassLoading` *naive*

In total there are 199,499 concrete methods representing 3,927,726 instructions. The maximum overhead for constructors represents 213,395 instructions. The maximum overhead for the methods represents 997,576 instructions. The total maximum overhead for method instructions is 30.83%. If we assume that every instruction makes 10 bytes (an over-approximation), the memory necessary to represent the bytecode increases from 37 Mib to 49 Mib. The total size of all classes (including not only the bytecode of methods, but also the constant pools, the attributes, etc.) increases from 160 Mib to 172 Mib which represents a maximum overhead of 7.5%.

In practice, as explained in Section V-D below, only 1.33% of classes need to be instrumented. Therefore, the overhead only represents less than 3000 instructions for constructors and less than 13300 instructions for methods. This represents an overhead of 159 Kib (0.42%).

The impact of MUSTI on the memory is low. In practice, new instructions add an overhead 0.42% which, in typical Java environments, represents less than 200 Kib.

D. RQ4: Vulnerable Code

Through this research question we aim at measuring the attack surface of the Java Class Library (JCL) for the invalid object initialization vulnerability. We search for vulnerable constructor methods and count them. To evaluate the number of vulnerable constructors, we developed a program based on Soot [30] to statically analyze the constructors and extract the permissions they check. The analysis first constructs a CHA-based [23] call graph starting from the constructors' methods. Then, it searches the call graph down to a depth of 6 for methods M checking for a permission (those are well-defined in the Java documentation [7], [1]). Similarly to previous work [16], we assume permissions are checked early in the call-graph. For every M, it performs a context-sensitive backward analysis to extract the string representing the permission which is checked.

In total, we identified 938 constructors checking for 36 different kinds of permissions in 353 classes (1.33% of the 26,610 classes of the JDK). Notable vulnerable constructors are found in classes such as `java.lang.ClassLoader`

(see Figure 1) where a permission is checked at a depth of 3, or in class `java.net.DatagramSocket` where a permission is checked at depth 4.

> The number of classes actually checking for a permission in a constructor is small (1.33%) compared to the total number of classes in the JDK. This information can be used to optimize the number of classes and methods to instrument to reduce the runtime and memory overheads.

VI. Limitations

A. Bytecode Length Limit

The JVM restricts the bytecode size for a method to be less than 65,236 bytes [11]. In our experiments, we have never seen a method with a bytecode size greater than 25,744 bytes [10]. An analyst could craft such a method to prevent our approach from updating the bytecode. However, such a huge size for a method can be trivially detected and trigger a red flag to stop a potential attack. Furthermore, while our current implementation updates the bytecode, it could be updated to append native code when the bytecode is interpreted. This would make this problem of the bytecode size insignificant. Furthermore, in the optimized version, MUSTI only instruments system classes whose constructors are checking for permissions. In that case, the size of classes crafted by the analyst would have no influence at all.

B. Field and Method Number Limit

The JVM restricts the number of methods to 65,535 and the number of fields to 65,535 [9]. Again, we have never seen classes with more than 1,260 methods [11] and 360 fields [12]. An analyst could craft such a class to prevent our approach from adding new fields. Nevertheless, this can be detected at runtime. Moreover, in the optimized version, MUSTI only instruments system classes. In that case, the number of fields and methods of classes crafted by the analyst would have no influence at all.

C. Other Attack Vectors

Our approach was designed to prevent invalid object initialization vulnerabilities. Other attacks based on buffer overflows, type confusion, confused deputy or other vulnerabilities which could also compromise the Java virtual machine sandbox are out of scope of this paper.

VII. Discussion

In this section we first discuss two other approaches to prevent the vulnerability in Section VII-A. We then demonstrate that bypassing our mitigation technique would require a vulnerability more powerful than an invalid object initialization vulnerability in Section VII-B.

[10] `sun.awt.X11.XKeysym: void <clinit>()`
[11] `com.sun.corba.se.impl.logging.ORBUtilSystemException`
[12] `com.sun.tools.classfile.Opcode`

A. Other Approaches

There are other approaches than the one described in this paper to prevent the exploitation of invalid object initialization vulnerabilities at runtime. We discuss them here and highlight their advantages and drawbacks.

1) Hard-code Checks in Source Code: One approach [6] which has already been partially implemented in the JCL is to explicitly hard-code checks for invalid object initialization instances in Java classes. While it prevents the vulnerability for being exploited in the updated class, the approach does not guarantee that all potentially vulnerable classes are protected. Furthermore, this approach adds $noise$ to the code which makes it harder to read and to maintain.

2) Patch the bytecode offline: The bytecode could be patched offline. That is, all $.class$ files could be modified to add code that will check for broken constructor chains. While this may reduce the overhead of loading classes, it also comes with limitations. First, only the bytecode of known classes can be modified and not the bytecode of classes created and loaded at runtime and loaded from the network. Second, the distribution of such code may break other programs which were not patched to support vulnerability check. Our approach, MUSTI, implements the checks directly in the JVM to make sure that the virtual machine state is consistent, i.e. that all classes are patched.

B. On the Possibility of Bypassing MUSTI

In this section, we informally argue that bypassing MUSTI would require the use of a vulnerability that can bypass the Java sandbox. That is to say, our approach works unless there is a critical vulnerability which would allow the analyst to bypass all security checks of the sandbox including MUSTI.

1) Private Field Bypass: An analyst could try to set the field $is\_initialized$ of an $Object$ instance to $true$ even if the object has not been correctly initialized. If the analyst can do that, he has a primitive to break the encapsulation of private fields of Java objects. He can thus directly modify the `private static volatile SecurityManager security` private field of the `System` class to disable all permission checks. He can thus bypass all restrictions of the Java sandbox, which is absurd.

2) Removing Permission Checks: An analyst could try to remove permission checks from system classes. If the analyst can do that it means he has a primitive to modify the bytecode for system classes which is a primitive more powerful than an exploit for an invalid object initialization. Thus, the analyst could define code in system classes to bypass all restrictions of the Java sandbox including MUSTI, which is absurd.

VIII. Related Work

Object Initialization. Rawtypes [29] is an approach based on a type system to enforce the proper initialization of object in Java. Both Rawtypes and MUSTI can guarantee that objects checking permissions in their constructors are properly initialized. Rawtypes relies on a static approach which requires manual work to annotate the code. On the contrary, our

approach, MUSTI, is dynamic and fully automated. Furthermore, we show that only 1.33% of the Java classes have to be instrumented when protecting the VM against a sandbox bypass. The consequence is that, in practice, MUSTI has a very low runtime and memory overhead. Rawtypes and MUSTI can also ensure that all objects are properly initialized. In that case, Rawtypes would require further manual work to add the adequate annotations and MUSTI would have a higher overhead since more classes are instrumented.

Freund et al. [25] have also developed a type system to ensure proper initialization of Java objects. However, the approach does not guarantee that partially objects cannot be accessed.

Java Security. Oh et al. [31] analyze CVE-2012-0507, a type confusion vulnerability, and explain how it has been used by malware.

Auriemma et al. [24] present techniques to bypass detection of known Java exploits by security tools. Techniques include serialization, splitting the exploit in multiple parts or leveraging multiple JVM.

Holzinger et al. [28] have studied public Java exploits. Their findings highlight that exploits leverage, among others, the following weaknesses of the Java platform: unauthorized use of restricted classes, arbitrary class loading and caller sensitivity. The paper presents the most vulnerable parts of Java but does not give any solution for preventing attacks based on the vulnerabilities.

Wang et al. [34] discuss simple techniques to detect Java exploits. One technique consists in disabling the security manager and run the suspicious Java code while checking if it tries to disable the security manager. If it does, there is a very high probability of it being a Java exploit. While most proof-of-concepts aim at disabling the security manager, not all practical attack actually need a security manager set to `null`.

Coker et al. [21] evaluates how the security manager is used in benign applications. Based on this knowledge, they devise two rules to prevent most of the exploits from working: the security manager cannot be changed if it has been set by the application and a class may not directly load a more privileged class if a security manager is set. This does not prevent malicious code from bypassing permission checks in constructors.

Holzinger et al. [27] presented an approach to remove shortcuts in stack-based access controls. While this improves the overall security of the JCL by making it much harder to have confused deputy attacks, it does not prevent attacks based on the vulnerability presented in this paper.

IX. CONCLUSION

In this paper we have presented an approach, MUSTI, to prevent the exploitation of invalid object initialization vulnerabilities. From a security point of view it is essential to protect against this kind of critical vulnerability since it may allow to completely bypass the Java sandbox. MUSTI successfully prevents exploits based on invalid object initialization vulnerabilities. In practice, our approach has a very low runtime overhead less than 0.5% and a very low memory overhead less than 0.45%.

APPENDIX A
SOFTWARE VERSIONS

For the sake of reproducibility, we list below the version of the programs/libraries/applications/files we used for our implementation and for the experiments.

- openjdk-8_8u144-b01-1.debian.tar.xz f0f94bd01397abdd966e64918bf3b350fc8c08b020-eeeaf386d2dc76ff8554a7 (sha256)
- openjdk-8_8u144-b01.orig.tar.gz e816e1a8e2fee6ce21335cd8159805bde8e04be1c5-8214037cf39950fba991e5 (sha256)
- Soot commit cdef52ed39e849565e60609328017fe4885bd3d7
- Java-ML version 0.1.7
- DaCapo version 9.12
- Android application a02fe87870ece6e4772db1445670cfc5f06cf7cd5f-646c457dac4eccb787e6be (sha256)

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under project CHARACTERIZE C17/IS/11693861 and by the University of Luxembourg under the VulFix project.

REFERENCES

[1] Accesscontroller. Oracle, https://docs.oracle.com/javase/8/docs/api/java/security/AccessController.html.
[2] Bug 1413562 - (cve-2017-3289) cve-2017-3289 openjdk: insecure class construction (hotspot, 8167104). Redhat https://bugzilla.redhat.com/show_bug.cgi?id=1413562.
[3] Excelsior jet - java virtual machine (jvm) and native code compiler. Excelsior, https://www.excelsiorjet.com/.
[4] J9 virtual machine (jvm). IBM, https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.win.70.doc/user/java_jvm.html.
[5] Lesson: Java applets. Oracle, https://docs.oracle.com/javase/tutorial/deployment/applet/.
[6] Obj11-j. be wary of letting constructors throw exceptions. https://wiki.sei.cmu.edu/confluence/display/java/OBJ11-J.+Be+wary+of+letting+constructors+throw+exceptions.
[7] Securitymanager. Oracle, https://docs.oracle.com/javase/8/docs/api/java/lang/SecurityManager.html.
[8] Vulnerability summary for cve-2017-3289. National Vulnerability Database https://nvd.nist.gov/vuln/detail/CVE-2017-3289.
[9] The java virtual machine specification, java se 7 edition: 4.1. the classfile structure. Oracle https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.1, 2013.
[10] The java virtual machine specification, java se 7 edition: 4.10.1. verification by type checking. Oracle https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.10.1, 2013.
[11] The java virtual machine specification, java se 7 edition: 4.9. constraints on java virtual machine code. Oracle https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.9, 2013.
[12] The final countdown for npapi. Google, https://blog.chromium.org/2014/11/the-final-countdown-for-npapi.html, 2014.
[13] Npapi plugins in firefox. Mozilla, https://blog.mozilla.org/futurereleases/2015/10/08/npapi-plugins-in-firefox/, 2015.
[14] Thomas Abeel, Yves Van de Peer, and Yvan Saeys. Java-ml: A machine learning library. *Journal of Machine Learning Research*, 10(Apr):931–934, 2009.

[15] Alexandre Bartel, Jacques Klein, and Yves Le Traon. Musti: Dynamic prevention of invalid object initialization attacks. Technical report, Université du Luxembourg, 2018.

[16] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 274–277. ACM, 2012.

[17] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.

[18] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. *IEEE Transactions on Software Engineering*, 40(6):617–632, 2014.

[19] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[20] Cristina Cifuentes, Nathan Keynes, John Gough, Diane Corney, Lin Gao, Manuel Valdiviezo, and Andrew Gross. Translating java into llvm ir to detect security vulnerabilities. In *LLVM Developer Meeting*, 2014.

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

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

[23] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[24] Donato Ferrante and Luigi Auriemma. Reloading java exploits. In *Hack in the Box*, 2014.

[25] Stephen N Freund and John C Mitchell. *A type system for object initialization in the Java bytecode language*, volume 33. ACM, 1998.

[26] LSD Research Group et al. Java and java virtual machine security, vulnerabilities and their exploitation techniques. In *Black Hat Briefings*, 2002.

[27] Philipp Holzinger, Ben Hermann, Johannes Lerch, Eric Bodden, and Mira Mezini. Hardening java's access control by abolishing implicit privilege elevation. In *2017 IEEE Symposium on Security and Privacy (Oakland S&P)*, 2017.

[28] Philipp Holzinger, Stephan Triller, Alexandre Bartel, and Eric Bodden. 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)*, 2016.

[29] Laurent Hubert, Thomas Jensen, Vincent Monfort, and David Pichardie. Enforcing secure object initialization in java. In *European Symposium on Research in Computer Security*, pages 101–115. Springer, 2010.

[30] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[31] Jeong Wook Oh. Recent java exploitation trends and malware. In *Black Hat USA*, 2012.

[32] Nathanael Paul and David Evans. Comparing java and .NET security: Lessons learned and missed. *computers & security*, 25(5):338–350, 2006.

[33] Jerome H Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 1974.

[34] Xinran Wang. An automatic analysis and detection tool for java exploits. In *Virus Bulletin*, 2013.