Bugfuscation

Alexandre Bartel

Umeå University, Umeå, Sweden
alexandre.bartel@cs.umu.se

Abstract. We introduce bugfuscation, a code obfuscation technique relying on bugs or vulnerabilities to hide part of the control flow of a target program. The technique has been evaluated on Java programs on the Java virtual machine and also affects Android's Dalvik virtual machine and the more modern Android RunTime ART. The approach bypasses automated static program verification and validation techniques such as state-of-the-art taint trackers. At least 95.6% of all OpenJDK versions from 1.6 to 21.0.4 and 71.6% of Android versions from version 2.3 to 15 contain the necessary vulnerability to bugfuscate Java or Android code.

1 Introduction

Code obfuscation is a technique used to make the code harder to understand. This means that it will take more time for an analyst to reverse engineer the code. Given a skilled reverse engineer and enough time it will, however, be possible to break the obfuscation. Despite this limitation, this technique is still useful to deploy because the fact that the code stays protected for a certain period of time might be beneficial to the authors of the software. For instance, if most copies of the software are being sold during the first weeks after the release, meaning the copy protection mechanism cannot be reverse engineered and broken during that time, it will benefit the company selling the software. As with any technology, obfuscation can also benefit malicious actors, such as malware writers who could, for instance, bypass code validation steps to push an application to an application store such as Google's Play store. While the exact process that is used internally at Google to vet Android applications is not publicly known, previous research has shown that automation [47] using static or dynamic analysis is widely used to triage incoming applications to flag suspicious applications which require further analysis and even, probably for the most suspicious of them, manual analysis. In this paper, we focus on a technique to hide control flow from static analyzers.

Obfuscating the control flow can be done, for instance, through probabilistic models to make it more time consuming to reconstruct the original, "simple", control flow [50]. This kind of approach, however, does not hide the control flow but makes it harder to reconstruct since the obfuscation technique adds multiple paths with the same semantic for a single original path. Attempts to hide parts of the control flow have also been investigated by the research community and are divided into two main categories. In the first category, covert channels are used to transfer information in the program using information from the program

runtime or the operating system, such as timing between threads [58]. In the second category, a bug or a vulnerability in the hardware layer is used to hide control flow. For example, ExSpectre [62] uses a flaw in the CPU microcode handling speculative execution of assembly instructions to hide part of the control flow. In both cases, the obfuscation works, i.e., it hides the control flow from tools. Our approach relies on software bugs or vulnerabilities below or at the application layer to hide control flows in a program. The approach has been evaluated on Java. More precisely, we show that bugs at the level of the Java Virtual Machine (JVM) or the Java Class Library (JCL) can be exploited by a malicious actor to hide control flow from analysis tools. The advantage of this technique is that obfuscation can be performed on a Java program without the use of dynamic class loading, native code, or packing, which are often red flags, that might automatically classify the application as suspicious. The approach is also independent of the underlying hardware.

State-of-the-art static analyzers such as Facebook's Infer or FlowDroid [1] focus on analyzing the code of an application and do not precisely model the software or hardware layers under the application software layer. Indeed, modelling these layers would increase the complexity of the tool significantly because many different configurations would have to be modelled [1] but would also lead to longer runtime and memory overhead on top of the high probability of generating even more false positives. Hence, because of the lack of precise modelling due to the scope of static analysis, it is unsurprising that it becomes very challenging for these tools to identify the hidden flows. The approach relies on bugs. Therefore, it is effective as long as new bugs detected by an attacker are not found and fixed. Alternatively, if bugs or vulnerabilities are known – publicly or only to the software vendor – it is only effective on systems where these bugs are not corrected. In systems where good security practices are enforced, bugs are quickly fixed. Surprisingly, recent work has shown that on Android systems, even publicly known Java vulnerabilities can remain unpatched for years [56]. One possible reason is that Java code is not considered as critical – maybe because Java code is sandboxed like native code on Android – and thus the patching process is slow or could even be non-existing, meaning that the code will not be patched actively but only passively when the vulnerable code will be replaced by a newer and patched version. Therefore, on certain systems such as Android, this gives plenty of time to a malicious actor to use the bugfuscation technique.

In this paper, we show that it is possible to write pure Java code with no native code, no class loading and no obvious obfuscation techniques such as string encryption, to hide control flows from static analyzers. While the approach could rely on different JVM or JCL vulnerability types to build its obfuscation primitives, in this paper we focus on type confusion vulnerabilities. We use these vulnerabilities to make static analyzers think that the real type of an object is A while in reality, i.e., at runtime, it is B. State-of-the-art static analyzers cannot know the real type because they model the Java code layer, and the type

[1] already modelling one configuration is near impossible and this even if only the JVM code is taken into account and not the operating system or other software layers

confusion happens at a lower layer in the JVM code, which is abstracted away by the analysis. Our contributions are the following:

- We present bugfuscation, a novel approach to hide control flow from Java static analyzers. The approach relies on type confusion vulnerabilities used to hide control flow through virtual calls. As far as we know, this approach has never been described in the literature.
- We implemented bugfuscation as a tool called Bugfu. The approach successfully impacts OpenJDK and the Android runtime, ART. All the seven state-of-the-art static taint analyzers such as Facebook's Infer and FlowDroid fail at detecting flows hidden with Bugfu.
- We evaluate the feasibility of such attacks in the real world. By analyzing known Java and Android vulnerabilities, we observed that 95.6% versions of OpenJDK from 1.6 to 21.0.4 and 71.6% of Android from version 2.3 to 15 contain the necessary vulnerability to bugfuscate Java or Android applications. The vulnerabilities have a lifetime of up to nine years which makes the obfuscation approach realistic, e.g., to bypass app. vetting mechanisms or for supply chain attacks.

2 Background

Static Taint Analyzers A taint analysis tracks data information in a program. Data generated by a *source* is tainted. When this tainted data is stored in a variable, the taint is propagated to the variable. A taint analysis could, for instance, track data generated from sources and makes sure the data does not leak out of the program through *sinks*. In Figure 1, with `getSecret()` as a source and `sendHttp()` as a sink, a taint analysis should find that the secret value s returned by the source at line 4 does not reach the sink at line 11 and thus that s does not leak out of the program. A static taint analysis models the program without executing it. It, e.g., builds a control flow graphs (CFG) to model the flow between statements of a single method or models the calling relationships between methods.

```
1  class Main {
2    public static void main(String[] args) {
3      A a = new A();
4      int s = getSecret();
5      I i = a;
6      i.m(s);
7  }}
8  interface I { void m(); }
9  class A implements I  { void m(int p1) { return; }; }
10 class B implements I  { void m(int p2) { return; }; }
11 class V {  void m(int p3) { sendHttp(p3); } }
```

Fig. 1: What method is called at line 6?

Main Approaches One approach consists in using IFDS, a framework used for interprocedural dataflow analysis which reduces it to a graph reachability problem [54]. This approach is used by tools such as FlowDroid [1]. Other taint analyzers such as Infer [8], rely on a technique called bi-abduction [9] and may combine sepration logic [55] with additional techniques [64,10,3] to be able to modularly analyze programs. Another approach is based on a form of type-based static analysis. This is the case of the Checker framework [20] where source and sink methods have to be annotated with @Tainted or @Untained tags. It is grounded in the theory of pluggable type systems [49] and the type qualifier system [25].

The flavor of static taint analysis a tool is using does not impact our approach. What does are the parts of the software stack which are taken (or more likely not taken) into consideration for analysis and, more precisely, how the tool models the underlying software layers. Some approaches, for instance, model the Java API based on the specifications or the API interface to evaluate how taints propagate. Based on this modeling, the analyzers could be sound in respect to the API specifications. However, this soundness is not guaranteed when there are bugs in the implementation of the API enabling to break the specifications. To the best of our knowledge, in the case of Java program analysis, all analyzers assume that the underlying VM is bug-free. We will see in the motivation example that this might result in the tools missing flows and thus missing information leaks.

3 Threat Model

In this paper we consider two scenarios for the attacker. In the first scenario, the attacker aims at bypassing vetting mechanisms based on program analysis to spread malicious code to a market such as Google's Play Store. In the second scenario, the attacker aims at bypassing vetting mechanisms based on program analysis and/or manual analysis to perform a supply chain attack and introduce malicious code such as a backdoor in a software project which could then be spread within a target (software or company) relying on this open source project.

We assume that the attacker targets a Java based application and/or library. The attacker does not control the Java or Dalvik virtual machine on which the application runs, but has full control over the application code in the first scenario and some control over the source code in the second.

4 Motivation

Most taint analyzers are not totally sound but soundy [38], i.e., they seek to maximize soundness while maintaining a reasonable balance between precision and scalability. Even if they are totally sound, they all rely on certain explicit and implicit assumptions on the software layers below the Java program. In the case of taint analyzers for Java, one assumption is that the underlying JVM,

implemented in C/C++, and the Java classes shipped with the JVM do not contain any bug.

With a type confusion, a reference to a type T_1 references an unrelated type T_2. In the case of the Java virtual machine, the VM thinks that the type of an object is T_1 while its real type is T_2. This situation is illustrated in Figure 2. An instance of A is created at line 2 and an instance of V is created at line 3 (see Figure 1 for the definition of A and V). We call their allocation sites, AS_1 and AS_2, respectively. At line 4, the type confusion is used to change the reference of a, of type A, to make it point to AS_2, an instance of type V. The virtual machine will still consider a as a reference to an instance of type A while the real type of the instance in memory is now type V. The secret key is put in `s` at line 5. The secret key is given as a parameter to method `m()` at line 6. Which method `m()` will be called? At runtime, method `V.m()` is called and leaks the secret through the sink `sendHttp()`. However, the static analyzer will conclude that there is no leak of the secret value. Indeed, using the call-graph generated by CHA, one concludes that the set of possible types for a is only $\{A\}$. Thus, the analyzer concludes that only `A.m()` can be called and thus that there is no leak because `A.m()` does not call the `sendHttp()` sink (see Figure 1).

```
1  public static void main(String[] args) {
2    A a = new A(); // AS1
3    V v = new V(); // AS2
4    a = useTypeConfusion(v);
5    int s = getSecret();
6    a.m(s);
7  }
```

Fig. 2: Generic type confusion: `useTypeConfusion()` allows to assign an object of type V to a reference of type A.

This example shows that fundamental algorithms, such as call-graph construction algorithms, can be manipulated by an attacker to hide specific method calls. The consequence is that all static analyzers relying on the results of these program abstractions will have an incomplete view of reality. In particular, some data flows can be hidden from the analyzer to prevent it from statically detecting chosen behaviors, such as malicious activity.

To better visualize all the necessary code to perform a type confusion, we show a concrete `typeConfusion` method in Figure 3. A concrete implementation varies depending on the vulnerability used to achieve type confusion. In this particular example, the implementation relies on CVE-2017-3272. In short, this vulnerability in the Java API implementation allows to update a field (`a`, line 4) with an object which does not have a compatible type (`b`, line 6). Eauvidoum and noise describe the vulnerability and its root cause [21].

```java
import java.util.concurrent.atomic.AtomicReferenceFieldUpdater;

public class M {
  protected volatile A a = null;
  class My {
    protected volatile V v = null;
  }

  // Returns 'v' as a valid A reference
  public static A useTypeConfusion(V v) {
    AtomicReferenceFieldUpdater updater =
    ↪  AtomicReferenceFieldUpdater.newUpdater(My.class, V.class, "v");
    M mini = new M();
    updater.set(mini, v); // type confusion happens here because of missing checks on 'v'
    ↪  in method set
    return mini.a;
  }
}
```

Fig. 3: Concrete implementation of the `typeConfusion` method based on CVE-2017-3272

5 Approach

As input, Bugfu takes the Java program to transform, the list of methods to bugfuscate and the vulnerabilities to use to hide the control flows. In the transformation step, the original program is automatically transformed to hide the methods in the list from static analyzers. For every method call `C.mc` to hide, a mirror method `mc` with the same signature is created in a new class `CMirror`. Then, the original call to `C.mc` is replaced by a call to `CMirror.mc`. A type confusion is also inserted so that at runtime the call to `CMirror.mc` is actually done on the instance of `C` of the original `C.mc` call. This will in effect redirect the call to the original `C.mc` method. We use the fact that the JVM does not check types at runtime for virtual calls done through the `virtualinvoke` instruction to actually redirect the call to `C.mc`. Therefore, the constraint is that the virtual tables of classes `C` and `CMirror` have their methods `mc` exactly at the same index. To make sure that this is the case, we count the number of methods in class `C` before `mc` and add the same number of methods in class `CMirror`. Thus, at runtime, `CMirror.mc` will be located at the same index in the virtual table of `CMirror` as `C.mc` is in the virtual table of `C`.

An example of the process is illustrated in Figure 4. The program to obfuscate is on the top of the figure. The program is designed to retrieve the list of all contacts, something typically done in an Android environment (line 4) and send this list to a remote host on the Internet (line 5). The method to hide is `ToObfuscate.sendInternet(String)`. This method's index in the virtual table is 2 (we do not consider the methods inherited from `java.lang.Object` to simplify) since there is only one non-static method.

Therefore, Bugfu creates a new `Mirror` class containing two methods: one dummy method `m1` and one mirror method `sendInternet` which instead of sending the `String` argument to a remote host through an HTTP request (bottom, line 9), does nothing or does something benign like printing the argument on the screen (`println` on line 18). The original method call (top, line 5) is replaced

```
1  public class ToBugfuscate {
2   public static void main(String[] args) {
3    ToObfuscate o = new ToObfuscate();
4    String contacts = o.getAllContacts();
5    o.sendInternet(contacts);
6   }
7   public void getAllContacts() {...}
8   public void sendInternet(String s) {
9    sendHttp(s);
10  }
11 }
```

```
1  public class ToBugfuscate {
2   public static Mirror
    ↪  typeConfusion(ToObfuscate t) {...}
3   public static void main(String[] args) {
4    Mirror m = new Mirror();
5    ToObfuscate o = new ToObfuscate();
6    m = typeConfusion(o);
7    String contacts = o.getAllContacts();
8    m.sendInternet(contacts);
9   }
10  public void getAllContacts() {...}
11  public void sendInternet(String s) {
12   sendHttp(s);
13  }
14 }
15 public class Mirror {
16  public void m1() { return; }
17  public void sendInternet(String s) {
18   println(s);
19  }
20 }
```

Fig. 4: Sample code (left) being bugfuscated (right).

by a call to `Mirror.sendInternet()` (bottom, lines 4 and 8). While one could think that the call is done on an instance of `Mirror`, the actual call at runtime is done on `o` the instance of `ToObfuscate`. This is possible because of the type confusion vulnerability leveraged to store a reference to `ToObfuscate`, `o` in the reference `m` of type `Mirror` (bottom, lines 2 and 6).

6 Evaluation

We first present the dataset of vulnerabilities we have used for the evaluation in Section 6.1. Then, we answer to the following research questions in Section 6.2 and in Section 6.3, respectively:

- **RQ1**: Can state-of-the-art tools detect bugfuscated control flows?
- **RQ2**: What are the characteristics of type confusion vulnerabilities?

6.1 Dataset

We collected type confusion vulnerabilities using the following methodology. We download all CVEs affecting OpenJDK from the period 2014-2024 from the National Institute of Standards and Technology (NIST)'s vulnerability database [39]. We only keep vulnerabilities for which the description mentions that it allows to compromise the security of the Java Virtual Machine, because type confusion vulnerabilities fall into this category. We try to find a publicly available proof-of-concept (PoC). If no PoC is found, we manually look at the source code of the patch to understand if the vulnerability is a type confusion or can be leveraged to trigger a type confusion. Out of the 290 CVEs we have collected, 75 have the potential for a type confusion. Out of these 75, we have found a PoC for five CVEs and identified two CVEs for which the patch indicates that the vulnerabilities can be used to perform a type confusion. Thus, in total, we have a set of

seven OpenJDK vulnerabilities: CVE-2014-0456 [42], CVE-2015-4843 [40], CVE-2016-3587 [41], CVE-2017-3272 [43], CVE-2018-2826 [44], CVE-2024-20919 [45] and CVE-2024-20921 [46].

When we manually analyze vulnerability patches, we only keep vulnerabilities for which we are highly confident they can lead to type confusion because, for instance, they impact the same software module as a vulnerability for which we do have a PoC. Therefore, we might have missed CVEs that could lead to type confusion vulnerabilities but for which our manual analysis was unconclusive. Our objective is not to obtain an exhaustive list of all vulnerabilities leading to type confusion but to illustrate that the approach described in this paper is of practical concern and not only a theoretical concept.

6.2 RQ1: State-of-the-art Static Taint Analyzers and Bugfuscated Control Flows

With this research question, we evaluate the capability of state-of-the-art Java static taint analyzers to detect tainted paths in bugfuscated Java programs. In the experiments, we evaluate seven tools chosen to reflect the diversity of academic and industrial static taint analysis tools: IBM AppScan [34] 2018, FlowDroid [1], Facebook's Infer, Sonar's SonarQube Cloud, the Checker Framework [20], Doop P/Taint [29], and Joana [30].

We evaluate these tools with the set of five vulnerabilities for which we have a PoC. For every vulnerability $V_i \in$ {CVE-2014-0456, CVE-2015-4843, CVE-2016-3587, CVE-2017-3272, CVE-2018-2826}, we developed an bugfuscated Java program JV_i, similar to the program in Figure 2, to try to hide the control flow leaking private information from static analyzers. Each program has the same source `getSecret()` (line 5), the same sink `sendHttp()` (represented on line 19 in Figure 1), but a different implementation of `useTypeConfusion` (line 4) depending on the vulnerability used. We test each program we develop on a vulnerable JDK version. They all successfully leverage the vulnerability to leak the private information at runtime. The seven static taint analyzers are able to find a leak on the five non-bugfuscated programs. We evaluated the seven static taint analyzers on the five Java obfugscated programs we developed. None of the tools is able to identify any of the bugfuscated leaks.

To better understand the impact on static analyzers which cannot detect bugfuscated paths, we conduct an additional study in which we experimentally compute and compare the call-graphs of an unbugfuscated program and its bugfuscated version. We bugfuscate our own Bugfu tool. The unobfuscated version contains about 1.6k nodes and 52k edges, while the bugfuscated version contains only 9 nodes and 57 edges. The drastic reduction in the size of the graph is explained by the choice of the method we obfuscated. This method is called from the main method and is the link from the main method to the core implementation of our tool. Once bugfuscated, the method call is hidden from static tools which results in a very small call graph, which only represents a tiny fraction of the possible execution flows.

6.3 RQ2: Lifetime of Java Type Confusion Vulnerabilities

To better characterize type confusion vulnerabilities, we conducted an empirical study on known type confusion vulnerabilities. To understand how many Java versions are impacted, we ran all five PoCs on Java versions 1.6 to Java 21. For the two vulnerabilities for which we do not have a PoC we find the original commit O_{com} introducing the vulnerability and assume all versions from O_{com} to the patched version are vulnerable. The results of this empirical study on OpenJDK versions are represented in Figure 5. The x-axis represents the publicly available OpenJDK versions from 1.6.0_01 released in May 2007 to OpenJDK 21.0.4 released in July 2024. Note that several versions – but not all – have "long term support" (LTS) which means that they still receive updates. At the time of writing, versions 1.8, 17 and 21 have LTS. The y-axis indicates how many of the seven vulnerabilities impact a version. We observe that all non-LTS versions (1.6, 1.7, 9, 10, 11, 12, 13, 14, 15, 16, 18, 19 and 20) are all impacted by at least one vulnerability. All LTS versions (1.8, 17, and 21) are also impacted at least by one vulnerability, except for the releases in which the latest vulnerabilities have been patched (1.8.0_401 and later, 17.0.10 and later, 21.0.2 and later). Out of the 183 OpenJDK releases analyzed in this study, 175 (95.6%) are impacted by at least one vulnerability, which can be leveraged to perform a type confusion attack.

The Android software stack also relies on Java and, more precisely, on part of OpenJDK's JCL in the latest versions. We checked for which vulnerabilities, the necessary code is also present in Android. Results are presented in Table 1. We have classified OpenJDK vulnerabilities in two groups. Vulnerabilities in the Hotspot group affect OpenJDK's Hotspot virtual machine implementation and cannot be present in Android since the Android VM implementation is not based on Hotspot. Vulnerabilities in the JCL group affect OpenJDK's Java Class Library and can be present in Android but only if the vulnerable code has been imported. Out of the seven vulnerabilities, four affect the JCL and three affect Hotspot. Out of the four affecting the JCL, two are imported to Android. There are also vulnerabilities affecting Android's implementation of the VM. We have searched for relevant research papers and CVEs to indentify type confusion vulnerabilities affecting the Android VM and have identified one vulnerability for which there is a PoC [7]. As far as we know, no CVE number has been assigned to this vulnerability. To understand how many Android versions are impacted, we analyzed 13 Android versions from 2.3 (2010) to 15 (September 2024). Results are presented in Figure 6. We observe that 10 Android releases (71.4%) are impacted by at least one vulnerability. We have written an Android application to make sure that the bugfuscation technique also works on Android. We leveraged CVE-2017-3272 which also affects Android. The Android application has been run on the Android emulator version 12L (API 32, released in March 2022) and type confusion can be legeraged successufuly to trigger a hidden method in a similar way as on the Java virtual machine.

The lifetime – the time between the first vulnerable version and the first patched version – of each OpenJDK vulnerability is represented in Figure 7. We

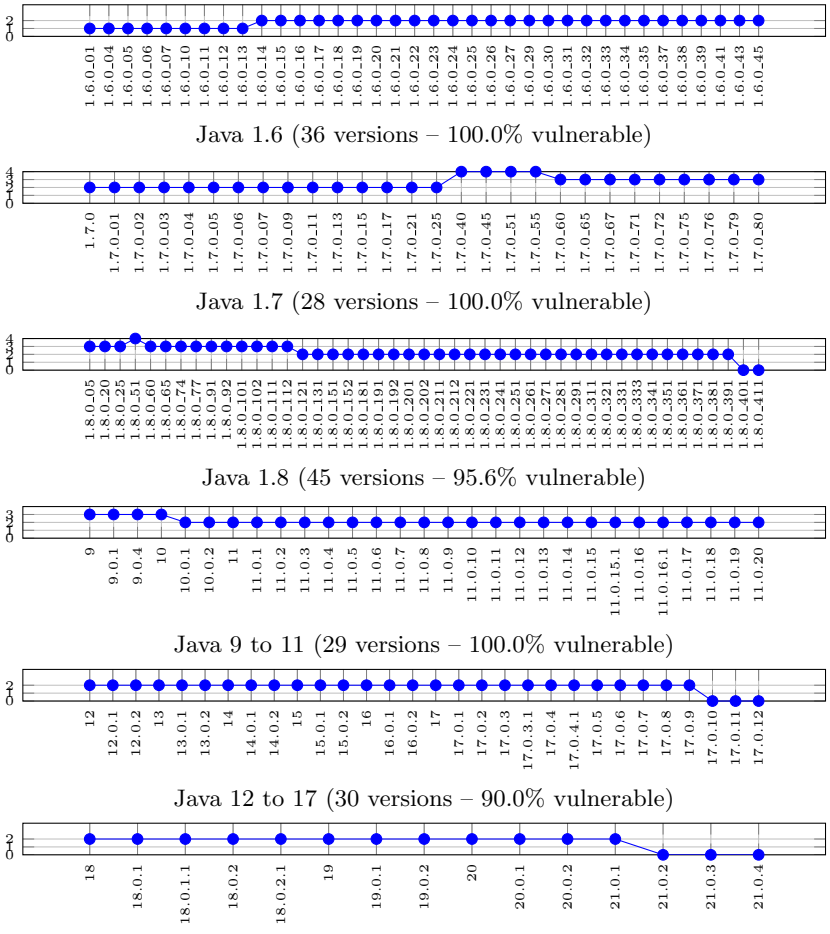

Fig. 5: Number of CVEs affecting Java 1.6 to 21

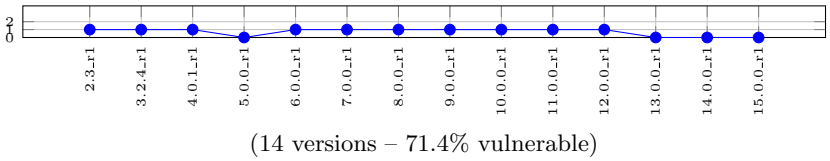

Fig. 6: Number of CVE's affecting Android 2 to 15

OpenJDK Vulnerability	Hotspot	Java Class Library	Affects Android
CVE-2014-0456	✓		
CVE-2015-4843		✓	✓
CVE-2016-3587		✓	
CVE-2017-3272		✓	✓
CVE-2018-2826		✓	
CVE-2024-20919	✓		
CVE-2024-20821	✓		

Table 1: OpenJDK vulnerabilities location in Hotspot or the JCL and their presence in Android.

observe that three vulnerabilities have a lifetime of about one year, one vulnerability has a lifetime of five years and three vulnerabilities have a lifetime of about nine years. The lifetime of Android vulnerabilities is illustrated in Figure 8. The figure represents one Android-specific vulnerability (Android-noCVE) and two vulnerabilities imported from OpenJDK. For these two vulnerabilities, the figure shows both the lifetype of the vulnerabilities in OpenJDK (CVE-2015-4843 and CVE-2017-3272) and their lifetime in Android (CVE-2015-4843-A and CVE-2017-3272-A). We observe that two vulnerabilities in Android have a lifetime of about a year (Android-noCVE and CVE-2015-4843-A) while one (CVE-2017-3272-A) has a lifetime of 6 years. We also observe that known OpenJDK vulnerabilities are imported in Android and not patched for a year after a patch is available for OpenJDK (for CVE-2015-4843-A) and for six years after a patch is available for OpenJDK (for CVE-2017-3272-A).

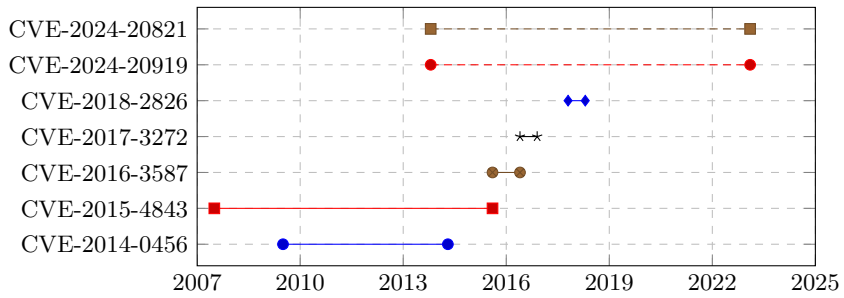

Fig. 7: Lifetime of OpenJDK vulnerabilities

7 Discussion

Only Type Confusion Vulnerabilities? Note that we focus on type confusion vulnerabilities in this paper, but other vulnerabilities might be used instead

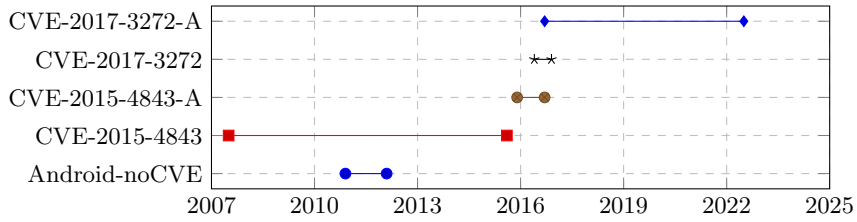

Fig. 8: Lifetime of Android vulnerabilities

in the approach. Actually, some vulnerabilities we consider are not type confusion vulnerabilities but can lead to a type confusion vulnerability. This is the case for instance with CVE-2015-4243 which is an integer overflow vulnerability. Thus, many memory corruption vulnerabilities and potentially other kinds of vulnerabilities in the Java Virtual Machine could be leveraged to achieve type confusion.

Solution: Model Known Vulnerabilities? One solution to improve static tools, could be to more precisely model known vulnerabilities. At the time of writing, however, there were no publicly known exploits for most of the hundred of undocumented Java vulnerabilities [19]. This means, that it is not yet possible to understand them, to model them and to detect Java code exploiting one of them to hide control flow without first reverse engineering them. This issue seems to be a major issue for static analyzers since (1) the developers of static analyzers often do not have the time nor the means to do the reverse engineering work for all vulnerabilities and (2) the software vendors often do not give enough information to understand and model vulnerabilities to be able to detect and compute the side effects of vulnerability exploitation. More research could be done to automatically understand the impact of known vulnerabilities and ease the modeling phase for static analysis tools.

Solution: Use Up-to-date Software? One could argue that using up-to-date software is the solution to prevent users from executing code exploiting known vulnerabilities. However, (1) users might not yet have made the switch to a newer version, (2) users might not want to update because of compatibility issues, (3) users might not be able to update because of cost constraints. For instance, Thomas et al. [59] have shown that more than 87% of Android devices are not using an up-to-date version of the firmware and are thus vulnerable to critical known vulnerabilities. Furthermore, for some Java virtual machine versions, security updates are only privately provided to customers and are not made public [48]. Last, but not least, an attacker could also find new vulnerabilities – unknown to the software vendor – in the latest release.

Solution: Checking Types? One could wonder why types are not checked at runtime, which would prevent type confusion vulnerabilities and thus limit the applicability of the obfuscation technique. The JVM contains a bytecode *verifier*, which ensures type safety of the loaded code. Because of these static guarantees, operations on types known in the bytecode are not checked at runtime. This mostly results in faster execution of the applications. Unfortunately, because of

the presence of type confusion vulnerabilities in almost all versions of the JVM, the guarantees on bytecode types are not guaranteed anymore.

However, should a static analysis detect that there is something wrong with types when the vulnerable code is called? Static analyses assume that the behavior of the JVM follows the specification and not the implementation. Therefore, if the Java API method `mvulnerable(A a, Object b)` is described in the specifications as *stores* `b` *into* `a` the static analysis computes the possible set of types for `b` and if there is a type incompatibility, will assume that an exception will be thrown and thus adds a control flow to the exception handler. In practice, if there is a type confusion vulnerability inside `mvulnerable`, the execution continues with the type inconsistency. However, this situation is not deemed possible by a static analysis tool which discards the corresponding flow.

Solution: Move Away from C/C++? The Java Virtual Machine is written mainly in C++. One could think that rewriting the Java virtual machine in another, memory safe, language such as Rust might fix this problem. While it might remove some vulnerabilities such as memory corruption which can lead to type confusion, it is not a full-proof solution. Indeed, some vulnerabilities are logic bugs and will still achieve type confusion even if the VM is writen entirely in a programming language such as Rust.

Read/Write Primitives Previous work [28] has shown that a type confusion vulnerability in the JVM yields *read* and *write* primitives. These could be used to, respectively, read and write bytes at an arbitrary position in memory, inject arbitrary code and redirect a virtual call to it. These primitives could have been used to bugfuscate applications. However, we have given priority to a simpler approach which is portable across all VMs that use the same representation for virtual tables. Nevertheless, it shows that the impact of such vulnerabilities is broad because they can be exploited in many ways.

Other Programming Languages While we have tested the approach for Java, it might work for other programming languages. One very close to Java is C#, which is also running on top of a virtual machine. Since the runtime is mostly written in C#, it is unlikely that a memory corruption is discovered in the Common Language Runtime (CLR). Nevertheless, a logic bug could in theory provide a primitive to bugfuscate the code. Many other languages such as Python, Eiffel, Scala can be used to generate Java bytecode. Thus, programs written in these languages could potentially also be bugfuscated depending on how the generated bytecode can be controlled through the compilation process. Last but not least, programs in C and C++ could also rely on type confusion vulnerabilities to hide control flow from static analyzers. These vulnerabilities [27] are often used to execute arbitrary code in web-browsers, for instance, and it is unclear if their life-time or cost [2] is interesting from an attacker's point of view to use them to bugfuscate code.

About a Supply Chain Attack A recent case of a backdoor injection attempt in OpenSSH through the *xz* dependency [26,17,31,4] also highlighted the time necessary for an attacker to gain trust and have code modification

[2] it terms of time spent to find the vulnerability

access right to the repository. If we consider the whole period from the social engineering aspect in 2021 to get access to the repository to the code modification to insert the backdoor in 2024, the attacker needed about 3-4 years. As we have seen, Java and Android vulnerabilities have a lifetime of up to nine years. This could give enough time to an attacker to leverage such vulnerability in this kind of attack.

Efficiency Our approach does not aim at competing against existing obfuscation techniques. It does not aim at being general nor efficient. However, depending on how it is implemented, the runtime overhead may be minimal and almost reduced to 0 because once the type confusion done (this can be done only once), each hidden method call costs the same as a normal method call. On the other hand, using obfuscation through the Java reflection API is much more costly at runtime, because the target method may have to be resolved every time reflection is used to call a method.

8 Related Work

Java code obfuscation Collberg et al. designed and implemented Kava, a tool to obfuscate Java programs [16]. They investigate techniques to obfuscate the control flow, such as opaque predicates or the insertion of irrelevant code, and techniques to obfuscate data structures. In addition, they describe metrics to evaluate each obfuscation transformation. Pizzolotto and Ceccato obfuscate Java applications by converting parts of the Java bytecode to native code [51]. Pizzolotto et al. [52] developed Oblive, a tool to help obfuscate Java and Android applications by converting part of the Java bytecode to native code and by hiding Java field values based on Xormasks [60]. Sakabe et al. developed an approach to reduce the precision of analyses computing points-to sets by modifiying classes so that they overload the same methods and guard the object creation code by opaque predicates [57]. Foket et al. also hide Java type information but through direct obfuscation of the type hierarchy [23,24,22]. [33] Chan and Yang overuse the same identifier for variable names, class names, method names, etc. and introduce techniques to introduce errors into the decompiled code to make it harder to reverse engineer [11]. Betchelder wrote a tool based on Soot to obfuscate Java applications [2]. The tool implements operator level obfuscation such as identifier renaming or arithmetic expression conversion as well as program structure obfuscation such as replacing `if` instructions by `try/catch` blocks. Proguard [36] is a tool to obfuscate Java code which implements string obfuscation variable renaming and other techniques.

Android Dalvik code obfuscation Android applications are shipped as Dalvik bytecode (compiled from Java or Kotlin code) and could also contain native code. Wong and Lie developed Tiro [63] a framework to detect common Android obfuscation techniques often used in malware. These include obfuscations manipulating the Dalvik representation of classes and methods through the use of native code. At runtime, the native code tempers the representation of Dalvik/Java classes or methods in memory to alter the control flow. Tiro's approach relies

on monitoring the native code of applications and thus would not directly detect bugfuscated flows in applications since no native code is required in our approach. Kovacheva developed a tool to obfuscate Android applications at the Dalvik level by adding calls to native wrappers, packing numerical variables, obfuscating strings or injecting bytecode instructions [35]. The main objective is to break disassemblers or decompilers.

Java Static Analysis for Security. Livshits et al. [37] developed an approach to find security vulnerabilities in Web applications. More specifically, it targets unchecked inputs, which can cause SQL injection vulnerabilities. Tripp et al. [61] developed Andromeda, a framework to perform static taint analysis. The framework improves precision and efficiency compared to previous approaches. Andromeda has been used in the IBM AppScan static taint analyzer [34] and in FlowDroid [1] a static taint analyzer for Android applications. They currently do not handle known vulnerabilities and thus could produce an unsound call-graph.

Java Static Analysis Improvement. Reif et al. [53] compare different call-graph algorithms implemented in Soot and IBM WALA in terms of soundness and show that many algorithms are already unsound. Major sources of unsoundness include reflection and new features added in Java 8 such as lambdas or method references. The consequence is that static analyzers built on top of Soot or WALA are unsound since they rely on call-graph with missing edges. We further show that vulnerabilities can be used to hide edges from the call-graph. Bonett et al. [5] used a mutation framework to find bugs in static taint analysis frameworks. They identified 13 flaws and fixed one of them with the help of the tool developers. As we do with this paper, the authors show the unsoundness of current static analysis tools. Unlike our work, they focus on bugs at the Java level, not on vulnerabilities.

Java Vulnerabilities. Java has a long history of security vulnerabilities. Most of the vulnerabilities have been mainly used to show that it is possible to escape the Java sandbox. In our approach, we show for the first time that they can be leveraged to obfuscate code. Holzinger et al. [32] analyze Java exploits ranging from confused deputy to deserialization issues to understand the weaknesses in the JVM in terms of weak implementation patterns or features. Dean et al. [18] describe multiple famous vulnerabilities enabling a Java sandbox escape. They also link vulnerabilities to weaknesses in the design methodology used in creating Java. For instance, the authors mention that Java lacks a formal semantics or a formal description of its type system. They further mention that this lack of formal description makes the bytecode verification very difficult. Furthermore, they describe the object initialization as being unnecessarily complex. As we have seen, many vulnerabilities are still being found in Java even in recent versions, including a vulnerability similar to one described in Dean et al.'s paper. This might indicate that not much has changed regarding the underlying Java design. Nevertheless, it seems that Oracle is taking steps to find vulnerabilities in Java [14]. Coker et al. [15] analyze recent Java vulnerabilities and show that adding restrictions on how to use the security manager and on how to augment privileges can prevent some exploits from running and preserve

backward compatibility with benign applications. While the technique prevents Java vulnerabilities from being exploited to gain elevated privileges or disable the sandbox, it will not detect or prevent vulnerabilities from being used to hide control flows as is done in our approach. Riom and Bartel [56] analyze how OpenJDK vulnerabilities can be imported into Android. They only consider vulnerabilities that can be triggered through untrusted data entering an Android application and reaching a vulnerable Java API. In our work, we consider Java vulnerabilities that applications themselves can use to hide control flows. Their analysis also indicates that OpenJDK vulnerabilities are not patched in Android and can stay unpatched for years after a patch is publicly available.

Finding Java Virtual Machine Bugs and Vulnerabilities Chen et al. introduce Classfuzz, a mutation based fuzzer to test the JVM [13]. The authors have used about 1500 class files from the Java Runtime Environment 7 as required initial seeds. The same mutated input is then fed to different implementations of the Java Virtual Machine. If the output of all these runs on the same input is different, a manual analysis is performed to identify potential bugs. Chen et al. developed ClassMing, [12] a mutation based fuzzer for the Java Virtual Machine. Using differential testing, 14 bugs have been identified. Bonnaventure et al. [6] developed Confuzzion, a generic fuzzer to find vulnerabilities in the Java Virtual Machine. They do not discover any new vulnerabilities, but show that Java programs triggering existing vulnerabilities can be generated from scratch in a few hours given a limited search space among the Java classes.

9 Conclusion

We concretely show that state-of-the-art static taint trackers for Java are unable to detect hidden control flows constructed based on type confusion vulnerabilities. These vulnerabilities are present in at least 95% of OpenJDK releases and 71.6% of Android releases and we find that they can have a lifetime of up to nine years. In some systems, such as Android, the known vulnerabilities imported from OpenJDK that we have analyzed were not quickly patched and we find that they can be present in the code up to six years after a patch is publicly available. These characteristics make them interesting targets for attackers to bugfuscate their code to bypass vetting mechanisms and target end-users by spreading through application stores or via a supply-chain attack.

Having a static analyzer that automatically detects all bugfuscated programs is not realistic since it would mean that it could automatically detect all vulnerabilities leading to type confusions, which is an open problem. Nevertheless, static analyzers could be shipped with a list of known vulnerabilities to at least detect when these known vulnerabilities are leveraged to hide control flows.

Tool Availability

Bugfu is available at this link:
`https://github.com/software-engineering-and-security/bugfu`.

Information about Tool Versions

IBM AppScan was sold to HCL Software in 2019. All vulnerabilities we used to evaluate the tool are from 2018 or before. We have contacted HCL Software, but did not get access to their version.FlowDroid: `https://github.com/secure-software-engineering/FlowDroid`, version 2.13. Infer: `https://github.com/facebook/infer`, version 1.2.0 for Linux x86_64. SonarQube Cloud: `https://www.sonarsource.com/products/sonarcloud/`, version 2025.Feb. Checker Framework: `https://github.com/typetools/checker-framework/` version 3.48.2. Doop P/Taint: `https://github.com/plast-lab/doop`, version 215c13c6. Joana: `https://github.com/joana-team/joana`, version c0687afb: this is the 'commit close to the latest commit' for which Joana compiles

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

1. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.
2. Michael Robert Batchelder. Java bytecode obfuscation. 2007.
3. Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W O'hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *International Conference on Computer Aided Verification*, pages 178–192. Springer, 2007.
4. Evan Boehs. Everything i know about the xz backdoor. `https://boehs.org/node/everything-i-know-about-the-xz-backdoor`, 2024. Accessed: 2024-09-09.
5. Richard Bonett, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. Discovering flaws in security-focused static analysis tools for android using systematic mutation. In *27th USENIX Security Symposium USENIX Security 18)*. USENIX Association, 2018.
6. William Bonnaventure, Ahmed Khanfir, Alexandre Bartel, Mike Papadakis, and Yves Le Traon. Confuzzion: A java virtual machine fuzzer for type confusion vulnerabilities. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 586–597. IEEE, 2021.
7. Jurriaan Bremer. Abusing dalvik beyond recognition. *Hack.lu, Luxembourg*, 2013.
8. Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.
9. Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–300, 2009.

10. Cristiano Calcagno, Dino Distefano, Peter W O'hearn, and Hongseok Yang. Footprint analysis: A shape analysis that discovers preconditions. In *Static Analysis: 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007. Proceedings 14*, pages 402–418. Springer, 2007.
11. Jien-Tsai Chan and Wuu Yang. Advanced obfuscation techniques for java bytecode. *Journal of systems and software*, 71(1-2):1–10, 2004.
12. Yuting Chen, Ting Su, and Zhendong Su. Deep differential testing of jvm implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1257–1268. IEEE, 2019.
13. Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of jvm implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99, 2016.
14. 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.
15. 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.
16. Christian Collberg. A taxonomy of obfuscating transformations. Technical report, Technical Report 148, 1997.
17. Russ Coxs. Timeline of the xz open source attack. `https://research.swtch.com/xz-timeline`, 2024. Accessed: 2024-09-09.
18. 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.
19. Debian. Information on source package openjdk-7. `https://security-tracker.debian.org/tracker/source-package/openjdk-7`. Accessed: 2024-09-09.
20. Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, 2011.
21. Ieue Eauvidoum and disk noise. Twenty years of escaping the java sandbox. *Phrack*, 2018.
22. Christophe Foket, Koen De Bosschere, and Bjorn De Sutter. Effective and efficient java-type obfuscation. *Software: Practice and Experience*, 50(2):136–160, 2020.
23. Christophe Foket, Bjorn De Sutter, Bart Coppens, and Koen De Bosschere. A novel obfuscation: class hierarchy flattening. In *Foundations and Practice of Security: 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers 5*, pages 194–210. Springer, 2013.
24. Christophe Foket, Bjorn De Sutter, and Koen De Bosschere. Pushing java type obfuscation to the limit. *IEEE Transactions on Dependable and Secure Computing*, 11(6):553–567, 2014.
25. Jeffrey S Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, 2002.
26. Andres Freund. backdoor in upstream xz/liblzma leading to ssh server compromise. `https://seclists.org/oss-sec/2024/q1/268`, 2024. Accessed: 2024-09-09.
27. Google. Type confusion in chrome lead to rce. Chromium, `https://bugs.chromium.org/p/chromium/issues/detail?id=722756`. Accessed: 2024-09-09.
28. Sudhakar Govindavajhala and Andrew W Appel. Using memory errors to attack a virtual machine. In *2003 Symposium on Security and Privacy, 2003.*, pages 154–165. IEEE, 2003.
29. Neville Grech and Yannis Smaragdakis. P/taint: Unified points-to and taint analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

30. Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.
31. Ben Hawkes. Openssh backdoors. `https://blog.isosceles.com/openssh-backdoors/`, 2024. Accessed: 2024-09-09.
32. 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.
33. Ting-Wei Hou, Hsiang-Yang Chen, and Ming-Hsiu Tsai. Three control flow obfuscation methods for java software. *IEE Proceedings-Software*, 153(2):80–86, 2006.
34. IBM. Appscan - application security. `https://www.ibm.com/security/application-security/appscan`. Accessed 2018-09-09.
35. Aleksandrina Kovacheva. Efficient code obfuscation for android. In *Advances in Information Technology: 6th International Conference, IAIT 2013, Bangkok, Thailand, December 12-13, 2013. Proceedings 6*, pages 104–119. Springer, 2013.
36. Eric Lafortune. Proguard. , `https://stuff.mit.edu/afs/sipb/project/android/sdk/android-sdk-linux/tools/proguard/docs/index.html`. Accessed 2018-09-09.
37. Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.
38. Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.
39. National Vulnerability Database (NIST). National institute of standards and technology - national vulnerability database. `https://nvd.nist.gov/`. Accessed: 2024-09-09.
40. National Vulnerability Database (NIST). Vulnerability summary for cve-2013-2423. `https://nvd.nist.gov/vuln/detail/CVE-2015-4843`. Accessed: 2024-09-09.
41. National Vulnerability Database (NIST). Vulnerability summary for cve-2013-2423. `https://nvd.nist.gov/vuln/detail/CVE-2016-3587`. Accessed: 2024-09-09.
42. National Vulnerability Database (NIST). Vulnerability summary for cve-2014-0456. `https://nvd.nist.gov/vuln/detail/CVE-2014-0456`. Accessed: 2024-09-09.
43. National Vulnerability Database (NIST). Vulnerability summary for cve-2017-3272. `https://nvd.nist.gov/vuln/detail/CVE-2017-3272`. Accessed: 2024-09-09.
44. National Vulnerability Database (NIST). Vulnerability summary for cve-2018-2826. `https://nvd.nist.gov/vuln/detail/CVE-2018-2826`. Accessed: 2024-09-09.
45. National Vulnerability Database (NIST). Vulnerability summary for cve-2024-20921. `https://nvd.nist.gov/vuln/detail/CVE-2024-20919`. Accessed: 2024-09-09.
46. National Vulnerability Database (NIST). Vulnerability summary for cve-2024-20921. `https://nvd.nist.gov/vuln/detail/CVE-2024-20921`. Accessed: 2024-09-09.
47. Jon Oberheide and Charlie Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 95:110, 2012.
48. Oracle. Java 7 information: Java se 7 end of public updates notice. `https://java.com/en/download/faq/java_7.xml`. Accessed: 2024-09-09.
49. Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, 2008.

50. Andre Pawlowski, Moritz Contag, and Thorsten Holz. Probfuscation: an obfuscation approach using probabilistic control flows. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*, pages 165–185. Springer, 2016.
51. Davide Pizzolotto and Mariano Ceccato. Obfuscating java programs by translating selected portions of bytecode to native libraries. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 40–49. IEEE, 2018.
52. Davide Pizzolotto, Roberto Fellin, and Mariano Ceccato. Oblive: seamless code obfuscation for java programs and android apps. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 629–633. IEEE, 2019.
53. Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. Systematic evaluation of the unsoundness of call graph construction algorithms for java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pages 107–112, 2018.
54. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.
55. John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th annual IEEE symposium on logic in computer science*, pages 55–74. IEEE, 2002.
56. Timothée Riom and Alexandre Bartel. An in-depth analysis of android's java class library: its evolution and security impact. In *2023 IEEE Secure Development Conference (SecDev)*, pages 133–144. IEEE, 2023.
57. Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Java obfuscation approaches to construct tamper-resistant object-oriented programs. *Information and Media Technologies*, 1(1):134–146, 2006.
58. Jon Stephens, Babak Yadegari, Christian Collberg, Saumya Debray, and Carlos Scheidegger. Probabilistic obfuscation through covert channels. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 243–257. IEEE, 2018.
59. Daniel R Thomas, Alastair R Beresford, and Andrew Rice. Security metrics for the android ecosystem. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 87–98. ACM, 2015.
60. Roberto Tiella and Mariano Ceccato. Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 182–192. IEEE, 2017.
61. Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 210–225. Springer, 2013.
62. Jack Wampler, Ian Martiny, and Eric Wustrow. Exspectre: Hiding malware in speculative execution. In *NDSS*, 2019.
63. Michelle Y Wong and David Lie. Tackling runtime-based obfuscation in android with {TIRO}. In *27th USENIX security symposium (USENIX security 18)*, pages 1247–1262, 2018.
64. Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *International Conference on Computer Aided Verification*, pages 385–398. Springer, 2008.