UNIVERSITÀ DEGLI STUDI DI MILANO
Facoltà di Scienze e Tecnologie
Dottorato di Ricerca in Informatica
XXXII ciclo
Settore scientifico INF/01

Static and dynamic analyses for protecting the Java software execution environment

Ph.D. Candidate
Stefano Cristalli

Tutor
Prof. Andrea Lanzi

Coordinatore del Corso di Dottorato
Prof. Paolo Boldi

Anno Accademico 2018/2019

Abstract

In my thesis, I present three projects on which I have worked during my Ph.D. studies. All of them focus on software protection in the Java environment with static and dynamic techniques for control-flow and data-dependency analysis. More specifically, the first two works are dedicated to the problem of deserialization of untrusted data in Java. In the first, I present a defense system that was designed for protecting the Java Virtual Machine, along with the results that were obtained. In the second, I present a recent research project that aims at automatic generation of deserialization attacks, to help identifying them and increasing protection. The last discussed work concerns another branch of software protection: the authentication on short-distance channels (or the lack thereof) in Android APKs. In said work, I present a tool that was built for automatically identifying the presence of high-level authentication in Android apps. I thoroughly discuss experiments, limitations and future work for all three projects, concluding with general principles that bring these works together, and can be applied when facing related security issues in high-level software protection.

Contents

Introduction

Several security problems affect high-level software environments. These problems can emerge regardless of security measures applied to the underlying systems. This is the case, for example, of problems arising from deserialization of untrusted data in several programming languages, such as Java, PHP, Python, and C#, where a technology with many legitimate uses (data serialization) is exploited to obtain unintended and malicious behavior in software. By leveraging object deserialization, attackers are able to chain pieces of benign software, leading to effects such as arbitrary code execution on the target system. Solving these problems while maintaining functionality is not trivial, because some of the causes that make the attack possible are needed key features of the serialization technology. Also, in the code chains mentioned above, it is hard to identify a single "culprit" that permits attacks; the malicious effects arise from the interaction of pieces of benign software, with no single point of failure. Another example of high-level security issue in software is the lack of authentication. We have an example of this in the Android environment, specifically when considering communication on short-distance channels, such as Bluetooth and WiFi-Direct. Authentication and encryption are enforced directly by the Android operating system, so that app developers can safely use standard APIs and benefit from these properties without reimplementing the underlying security protocols. However, authentication on the channel is not sufficient: in particular, it offers no guarantees on high-level authentication, the implementation of which is left to app developers. Experimental evaluation shows a lack of awareness of this problem in the community of Android app developers, at both amateur and professional levels. Again, we face an issue with difficult resolution: with the lack of APIs, developers must implement authentication on a per-app basis without a standardized approach, and awareness is difficult to increase without changes in the Android documentation.

These issues share common traits:

- They apply to Java-based environments.
- They are not caused by a specific low-level bug in code, or a single user's mistake; instead, they impact high-level security scenarios, with multiple causes and actors.
- They are related to anomalies in the normal execution (control flow) of

software.

Based on these unifying traits, we commit to the following **research statement** for the work presented in this thesis:

We wish to study detection mechanisms, solutions and mitigations for high-level security problems in Java-based environment by applying anomaly detection techniques, control flow and data flow analyses. We try to approach complex problems which have not yet been completely solved at the time of writing, and perform a comprehensive study on their causes and possible defenses.

As we state, a possible way to mitigate such security problems is the use of static and dynamic program analysis techniques. By inspecting either the source code, or the state of the software environment at runtime, it becomes possible to infer the presence of vulnerabilities in software, or the event of an ongoing attack during software execution.

Static techniques analyze unchanging information such as the source code of the software or the one of its environment. They are helpful for modelling control-flow information as well as data-dependency information. Dynamic techniques analyze volatile information at runtime, concerning the state of the software execution environment. By monitoring data, they are helpful for detecting anomalies in software execution.

In this thesis, I applied both static and dynamic analysis techniques to the resolution of the two security issues mentioned above:

1. The study of security problems related to deserialization of untrusted data in the Java Virtual Machine, and approaches towards their resolution
2. The study of problems related to the lack of high-level app-to-app authentication on short-distance communication channels in Android, and a system for detecting authentication in Android apps

My thesis is consequently divided in two major parts, each treating a separate problem. The main underlying topic is the use of the aforementioned techniques for software protection in Java environments.

Even though the analyzed problems have a high-level component which can be abstracted from the specific programming language and environment, we chose to focus on Java, tailoring our analysis and practical solutions to said language. Given its widespread adoption in both commercial and non commercial projects, and the variety of its applications (desktop software, web services, Android apps), in our thinking Java represents the ideal candidate for research on high-level software protection. The parts of our research that do not specifically rely on Java features (for example, the threat model of *CATCH* in Android) can be applied to other technologies; the more language-specific or architecture-specific parts (for instance, modifying the JVM to counter deserialization attacks) could also be adapted with additional technical work, but reusing the conceptual framework.

Part I is dedicated to "Deserialization of untrusted data in the Java Virtual Machine". After an introduction of the problem, I present the history of the vulnerability, along with the background concepts necessary to understand it. Afterwards, I present one research on the problem that I co-authored during my Ph.D., which describes a defense system to effectively mitigate this security problem by modification of the Java Virtual Machine. After a brief overview, I present the architecture of the system, then I proceed describing the technical details of its implementation. I conclude with experimental results, related work, and what future improvements we have planned. The last sections of Part I are dedicated to another research project, on which I have worked during my Ph.D., which aims at automating the process of discovery of new vulnerabilities based on deserialization of untrusted data, using static control-flow and data-dependency techniques. I follow the same structure for the presentation of this work.

Part II is dedicated to "High-level authentication on short-distance channels in Android". After introducing the problem, I present a third project completed during my Ph.D., in which the other authors and I present a system for detecting authentication on short-distance channels (or the lack thereof) in Android APKs. The presentation starts with a section for background concepts, followed by an overview of the system. Implementation details follow, before the part on experimental evaluation. Afterwards, I present two practical case studies on the problem, before moving on to the related work. A discussion on limitations and future work is presented last, with some concluding considerations.

The last section summarizes the work I presented in my thesis, in an effort to present general principles that can be applied when designing systems for the broad category of problems that is discussed, with specific insight on the control-flow and data-dependency analyses that are leveraged in the presented works.

Part I: Deserialization of untrusted data in the Java Virtual Machine

1.1 Problem introduction

Deserialization of untrusted data is a cause of security problems in many programming languages [17]. In Java, it might lead to remote code execution (RCE) or denial of service (DOS) attacks [59]. Even though it is easy to check whether preconditions for this type of attack exist in an application (that is, deserialization performed on user-controlled data), designing and carrying out a real attack is a hard task, due to the complexity of creating the attack payload. In order to exploit this type of vulnerability, an attacker has to create a custom instance of a chosen serializable class which redefines the `readObject` method. The object is then serialized and sent to an application which will deserialize it, causing an invocation of `readObject` and triggering the attacker's payload. Since the attacker has complete control on the deserialized data, he can choose among all the Java classes present in the target application classpath, and manually compose them by using different techniques (e.g., wrapping instances in serialized fields, using reflection), and create an execution path that forces the deserialization process towards a specific target (e.g., execution of a dangerous method with input chosen by the attacker). There are several public exploits [28] that show the impact of the attack on real Java frameworks, such as JBoss and Jenkins, which are based on several common Java libraries, such as Oracle JRE 1.7, Apache Commons Collection 3 and 4, Apache Commons BeanUtils, Spring Beans/Core 4.x and Groovy 2.3.x.

We give a brief introduction to the problem and the related background concepts in this section, and then proceed to illustrate our research contribute on the topic, in two distinct but related works:

1. In Section 1.3, we present a defense mechanism integrated in the JVM for runtime detection and protection against attacks based on deserialization of untrusted data.

2. In Section 1.4, we describe the steps we have made towards automated discovery of new malicious deserialization payloads in libraries, with the aim of developing tools to aid developers, uncovering potential bugs and increasing software quality.

We adhere to our research statement by treating a problem that has multiple high-level causes:

- The deserialization architecture in the Java Virtual Machine is vulnerable by design to the dangers of processing untrusted data, as a direct consequence of the high customization allowed in custom deserialization behavior of classes
- Software libraries leverage object deserialization to achieve key features
- Although no specific class is generally insecure, with evident security bugs, the interaction of deserialization code in different classes allows an attacker to obtain malicious behaviour where such classes are included in the classpath

Given the impossibility of performing a complete rework of the existent mechanism, or removing deserialization altogether, we treat the issue by designing detection and protection techniques, based on control flow and data flow analysis. The analysis of anomalies in the control flow allows us to construct a protection for the Java runtime environment, effectively enabling the detection and blocking of both known and novel deserialization exploits. Data flow analysis techniques applied to software libraries reveal the possibility of semi-automated discovery and validation of deserialization payloads, greatly reducing the amount of manual analysis required to analyze and secure code.

1.2 Background

In this section I describe background concepts for understanding the security problems with deserialization of untrusted data in Java. In particular, I briefly describe the Java Virtual machine and the HotSpot JVM's interpreter and compiler. Then I present the mechanisms of reflection and object deserialization in Java, and how the latter can be used to obtain malicious side effects when applied to untrusted data.

1.2.1 Java Virtual Machine

Java code runs on a virtual machine (the Java Virtual Machine, or JVM), which executes Java low-level instructions (bytecodes) on the host system. Several implementations for the JVM exist; in our research we focus on the HotSpot JVM, implemented by Oracle and used in both Oracle JDK and OpenJDK products. Since Java is an interpreted language, each JVM runs an interpreter, responsible

for translating Java bytecodes into machine instructions for the host's architecture as programs are executed.

The *template interpreter* is the current interpreter in use in the HotSpot JVM. The HotSpot runtime generates an interpreter in memory at the virtual machine startup using the information in the TemplateTable (a structure containing *templates*, assembly code corresponding to each bytecode). The TemplateTable defines the templates and provides accessor functions to get the template for a given bytecode [48].

In order to optimize performance, the JVM can compile some of the Java bytecodes into native code. The HotSpot JVM includes two Just-In-Time (JIT) compilers, C1 and C2, responsible for code optimization at runtime. C1, the *client compiler*, is optimized for compilation speed while C2, the *server compiler*, is optimized for maximal performance of the generated code. The HotSpot JVM constantly analyzes the code as it runs to detect the critical parts that are executed often (called *hot spots*, from which the JVM gets its name), which are then compiled into native code [30].

1.2.2 Java Technologies

Java reflection

Java allows reflection via a set of API calls. Reflective code can be used for various purposes, such as inspecting methods in classes and calling them dynamically. For example, a program could use reflection on a generic class instance `i` to check whether its class has a public method named `doSomething`, and invoke it on `i` in such case.

Java object serialization

Serialization is the process of encoding objects into a stream of bytes, while deserialization is the opposite operation. In Java, serialization is used mainly for lightweight persistence, network data transfer via sockets, and Java Remote Method Invocation (Java RMI) [50]. Java deserialization is performed by the class `java.io.ObjectInputStream`, and in particular by its method `readObject`. A class is suitable for serialization/deserialization if the following requirements are satisfied [51]: (1) the class implements the interface `java.io.Serializable`, (2) the class has access to the no-argument constructor of its first non-serializable superclass.

A class `C` can specify custom behavior for deserialization by defining a `private void readObject` method. If present, such method is called when an object of type `C` is deserialized. Other methods can be defined to control deserialization:

- `writeObject` is used to specify what information is written to the output stream when an object is serialized
- `writeReplace` allows a class to nominate a replacement object to be written to the stream

Listing 1.1: readObject in java.util.PriorityQueue

```
private void readObject(java.io.ObjectInputStream s)
  throws java.io.IOException,
         ClassNotFoundException {
  // Read in size, and any hidden stuff
  s.defaultReadObject();

  // Read in (and discard) array length
  s.readInt();

  queue = new Object[size];

  // Read in all elements.
  for (int i = 0; i < size; i++)
    queue[i] = s.readObject();

  heapify();
}
```

- `readResolve` allows a class to designate a replacement for the object just read from the stream

As an example of custom behavior in deserialization, in listing 1.1 we show the custom `readObject` method in class `java.util.PriorityQueue<?>`, which defines both `writeObject` and `readObject` to handle serialization/deserialization of the elements of the priority queue. We can see that queue elements are read from the `ObjectInputStream` one by one, by calling `readObject` multiple times, and then the function `heapify` is called in the end.

1.2.3 Vulnerability example

The code reported in listing 1.1 does not contain evident vulnerabilities. However, there is a security problem that could potentially arise from this mechanism: function calls defined inside `readObject` generally operate on data read from the stream, and such data can be controlled by an attacker. In such a context, an attacker can craft nested class objects in the deserialization input stream and define a sequence of method calls that end up executing dangerous operations at the operating system level, such as filesystem activities, command execution, etc. Chains of method invocations that lead to arbitrary command execution have been identified in different sets of classes in various libraries [28]. In general, it is hard to ensure that such gadgets do not exists in a given set of Java classes, due to the complexity in which their methods can be composed to create a valid execution.

In summary, three constraints need to be satisfied in order to obtain a successful attack on a Java application: (1) the attacker needs to define his own

invocation sequence by starting from a serializable class that redefines `readObject`; (2) to obtain malicious behavior, the attacker has to find a path that starts from the deserialized class and reaches the invocation of one or more desired methods; (3) all the classes considered in the attack execution path must be present in the application's classpath.

To give an example of a real attack, we present some code that shows how an attacker can pilot a deserialization process and execute a dangerous native method. In listing 1.2 we report the code for functions `heapify`, `siftDown` and `siftDownUsingComparator` of class `java.util.PriorityQueue`. In listings 1.3 and 1.4 we show methods `compare` of class `TransformingComparator` and method `transform` of `InvokerTransformer`, from library Apache Commons Collections 4. Listing 1.5 shows an hypothetical class for wrapping a system command.

Listing 1.2: heapify and siftDownUsingComparator in PriorityQueue

```
private void heapify() {
  for (int i = (size >>> 1) - 1; i >= 0; i--)
    siftDown(i, (E) queue[i]);
}

private void siftDown(int k, E x) {
  if (comparator != null)
    siftDownUsingComparator(k, x);
  else
    siftDownComparable(k, x);
}

private void siftDownUsingComparator(int k, E x) {
  int half = size >>> 1;
  while (k < half) {
    int child = (k << 1) + 1;
    Object c = queue[child];
    int right = child + 1;
    if (right < size && comparator.compare((E) c, (E) queue[right]) > 0)
    c = queue[child = right];
    if (comparator.compare(x, (E) c) <= 0)
    break;
    queue[k] = c;
    k = child;
  }
  queue[k] = x;
}
```

Listing 1.3: TransformingComparator.compare

```
  public int compare(final I obj1, final I obj2) { final O value1 =
    this.transformer.transform(obj1); final O value2 =
    this.transformer.transform(obj2); return
    this.decorated.compare(value1, value2); }
```

Listing 1.4: InvokerTransformer.transform

```
public O transform(final Object input) {
  if (input == null) return null;
  try {
    final Class<?> cls = input.getClass();
    final Method method = cls.getMethod(iMethodName, iParamTypes);
    return (O) method.invoke(input, iArgs);
  ...
}
```

Listing 1.5: Command class

```
public class Command implements Serializable {
  private String command;

  public Command(String command) {
    this.command = command;
  }

  public void execute() throws IOException {
    Runtime.getRuntime().exec(command);
  }
}
```

Listing 1.6: Sample payload

```
final InvokerTransformer transformer =
  new InvokerTransformer("execute", new Class[0], new Object[0]);

final PriorityQueue<Object> queue =
  new PriorityQueue<Object>(2, new TransformingComparator(transformer));

queue.add(1);
queue.add(new Command("rm -f importantFile"));
```

Now, suppose an attacker created and serialized an object as shown in listing 1.6. When this object is deserialized, the first method invoked after reading all the data from the priority queue is `heapify` as defined in the source code; then `siftDownUsingComparator` is called (via `siftDown`), which uses the com-

parator provided by the attacker into the serialized object, in this case a `TransformerComparator`, for comparing the queue elements. The `compare` function in `TransformerComparator` uses the field `transformer`, provided by the attacker, and calls its `transform` function on the objects being compared. `InvokerTransformer` uses reflection to call the method with name equal to its field `iMethodName` on `input`. The reflection in this case helps the attacker to invoke methods of generic classes; by crafting the deserialization input, the attacker is able to invoke method `execute` on an instance of the `Command` class with controlled parameters and execute arbitrary commands. In listing 1.7 we report the stack trace collected at the execution of `Runtime.exec`, which contains all the Java methods invoked during the malicious deserialization event.

Listing 1.7: Stack trace of sample attack payload

```
1 Runtime.exec
2 Command.execute
3 Method.invoke
4 InvokerTransformer.transform
5 TransformingComparator.compare
6 PriorityQueue.siftDownUsingComparator
7 PriorityQueue.heapify
8 PriorityQueue.readObject
```

The attack vector described in this section is based on payload "CommonsCollections2" from the ysoserial repository, used in real attacks. The main difference with the original version is that no class like `Command`, which was introduced for the sake of simplicity, is generally available in the classpath. The real attack vector uses a specific method chain that leverages dynamic class loading to pass from reflective method `invoke` to an execution of `Runtime.exec` with controlled input.

1.3 Trusted Execution Path For Protecting Java Applications Against Deserialization of Untrusted Data

The following sections describe a novel defense approach for the JVM against deserialization attacks, which I designed together with the other authors of the paper [16] published at RAID 2018.

1.3.1 Introduction

The main generally recognized defense against this issue is a whitelist/blacklist approach that allows only certain classes to be deserialized [54]. While blacklist approaches are very effective on known attacks, they cannot recognize novel exploits. Whitelists, on the other hand, suffer from one fundamental problem: the approach is based on static analysis (e.g. Look-Ahead Java Deserialization [54]) that processes the deserialization data input before the deserialization process has been executed. In such a context, static analysis fails to detect some attack vectors when, for example, the attacker uses reflection [34] or when he is able to dynamically load classes at runtime [28]. Our method dynamically tracks the execution path during the deserialization events.

Extracting such information from the Java execution model context is very difficult. In particular, we need to deal with several challenges due to the dynamic loading of Java classes at runtime, the JIT compilation mechanism and the native code instrumentation. Our proposed dynamic technique operates in this direction and is able to precisely reconstruct the dynamic execution path of object deserializations, and consequently mitigate the entire spectrum of the attacks based on deserialization of untrusted data.

More in detail, we propose a novel dynamic approach to protect Java applications against deserialization of untrusted data attacks. Our system is completely automatic and it is based on two phases: (1) training phase (2) detection phase. During the learning phase, the system collects important information about the behavior of benign deserialization processes and constructs the precise execution path in form of a collection of invoked Java methods (stack traces). In the second phase, the system runs a lightweight sandbox embedded inside the Java Virtual Machine, that acts during the deserialization process, and is able to ensure that only trusted execution paths are executed. Our tool is very flexible, and can be applied out of the box to protect any Java application. Its false positive ratio can be tuned according to the application behavior and to the desired level of protection. Our experiments, performed on two popular Java applications, JBoss and Jenkins, show the effectiveness and efficiency of our system.

To summarize, we make the following contributions:

- We design an approach to mitigate the problems of deserialization of untrusted data in the Java environment, based on the enforcement of precise

execution paths an agnostic of any variable that is external to the Java Virtual Machine (e.g., the operating system). We tackle several challenges regarding the extraction of dynamic information from the Java execution model such as: JIT compilation, runtime loading of the Java classes and native code instrumentation.

- We design a lightweight sandboxing system for the Java environment that is able to limit the attacker's actions and mitigate the attacks by using information from benign stack traces. Such a sandbox is transparent to Java applications and can be tuned according to a specific desired behavior.
- We perform an experimental evaluation on two real-world Java application framework: JBoss and Jenkins, and we show that our system is able to automatically extract detailed information about object deserialization and perform a precise detection. We also analyze the limitations of our system against new types of deserialization attacks.

We have presented background information related to Java deserialization technologies along with our threat model in Section 1.2. In Section 1.3.2, we discuss the principles of our defensive mechanism. Section 1.3.3 describes the architecture and details of the system. Experimental evaluation of our tool with various Java applications is discussed in Section 1.3.4. Related work is presented in Section 1.3.5. We provide a discussion of how our current implementation can be extended to handle other sophisticated deserialization attack techniques, such as data attacks, in Section 1.3.6.

1.3.2 Overview

We now briefly describe a high-level overview of our approach. In our work, we design an application centric model based on stack trace objects. More precisely, the stack trace structure is defined as a sequence (stack) of n objects, each of them is represented by one of the Java methods invoked during the deserialization process. The first element (entry point of the stack trace), is the first class that invokes the `readObject` method, while the last (exit point) consists of a native method call. It is important to note that one stack trace is always associated with only one native method invocation, and vice versa. In our detection model, we consider only stack trace associated with native methods that interacts with the operating system (e.g., process, filesystem and network activities). By defining which stack trace a particular deserialization event can invoke, it is possible to restrict the attack surface, mitigating the attack itself. The information about which stack trace can be invoked will be part of the sandbox policy, along with other information (e.g., native call) that is used to determine the behavior of the deserialization process.

In order to define the legitimate deserialization behavior in terms of execution path, we need to dynamically collect the stack trace for a monitored application by providing the appropriate input set to stimulate the deserialization process. During this phase, called constructing phase, the system collects

the entire observed benign stack trace related to deserialization events. This task is performed by dynamically monitoring a Java application at the interpreter/compiler level in the JVM. Extracting such dynamic information and constructing the execution path is a complex task due to the nature of the Java execution model, which includes JIT compiled code, and class loading at runtime. Static analysis cannot be used in such a context since the complete execution sequence of the Java methods is only known at runtime. Moreover such execution model is exacerbated by the use of Java reflection, which complicates static analysis itself. For example, a single dynamic method invocation that uses reflection could in principle invoke any method in the currently loaded classes, resulting in an over-approximate call graph and consequently enlarging the attack surface [34].

Once the system has established the benign stack traces, our framework can enter in the running phase, where it performs the detection task. At the starting point, our system loads a specific set of policies for each application derived from the constructing phase. Each set of policies is characterized by three elements derived from the stack trace collection: (1) entry Java class point, (2) invocation sequence of Java methods, (3) invocation of native method. The first element is the entry point that represents the Java class that is deserialized. All the deserialization operations for a specific application must start from a Java class observed during the training phase, any other invocation of `readObject` from any other class is blocked. The second element is the order of the sequence of method invocations. Such a sequence has been observed during the constructing phase and it must match the actual sequence at runtime, any deviation from its order will stop the deserialization process. The third element is the native call associated with the specific stack trace. Such native call will be checked by our framework, and any invoked native method that is not defined into the permitted set will trigger an alarm, causing the deserialization process to be terminated.

Although based on fine-grained policies, our approach is very flexible. In fact, our system can be tuned based on the level of information granularity that we want to use for detecting the attack. More specifically, we can configure the length of the extracted stack traces, and restrict or enlarge the attack surface. Acting on a subset of the entire stack traces sequence has a considerable advantage: by not checking all the sequence of the Java methods, the system can lower the false positives while maintaining a good precision in detecting the attacks as showed in the evaluation Section.

Threat Model

Our threat model considers an attacker who is able to exploit (either locally or remotely) an object deserialization on untrusted and user-controlled data inside a Java application running on the machine, and execute arbitrary method calls on classes present in the Java classpath. The attacker has full control on serialized data, as well as complete knowledge on the classes defined in the application classpath and their source code. We assume that the machine is

uncompromised when our defensive mechanism is loaded. We consider the Java Virtual Machine execution environment trusted and we assume that the attacker cannot compromise it by exploiting vulnerabilities such as memory errors.

1.3.3 System design and implementation

In this section we present the implementation details of our protection system. We describe the architectural overview and the challenges that we faced for dynamically tracing Java applications execution path.

Architectural overview

The principles provided in the previous section show how our approach can be used for extracting stack trace related to a deserialization event. There are two main requirements that need to be satisfied for our detection system: (R1) for any stimulated event, the system needs to be able to precisely monitor the execution path of the deserialization process, so that no relevant execution is missed; (R2) the monitor component should not cause a high overhead. From an architectural point of view our system is split into two main high-level components: (1) a component, that is in charge of dynamically analyzing Java applications and extracting the precise execution path in terms of stack traces, and (2) a lightweight sandbox component that monitors applications at runtime and blocks incoming attacks, based on the rules derived by the constructing phase.

More in particular, in the constructing phase, our system intercepts all the native methods invoked by the application; for each invocation, it inspects the corresponding stack trace backwards, until it reaches the deserialization entry point (i.e., a call to `readObject`), and then it extracts the execution path. In case the `readObject` is not found on the stack we assume that the native call is invoked in a different context than deserialization one and the system discards the results. It is important to note that the presence of the invocation of `readObject` method on the stack cannot be tampered by an attacker since deserialization of untrusted data attack does not allow to directly write on the stack. This information is then saved into a persistent storage called sandbox policy, which will constitute the baseline for detecting malicious behavior.

Afterwards during the detection phase, when user input triggers deserialization event, the system performs only one check according to the sandbox policy: when a native method is invoked by the application, the system intercepts it and checks whether the entire stack trace executed has been already observed in the learning phase. For this check the system maintains a memory structure in the form of a hash table that contains only the execution paths that are allowed for the applications. The keys of the hash table are strings composed by the methods signatures present in each benign stack trace. Since a sequence of method signatures uniquely identifies a specific stack trace (by definition), the risk of collisions is very low, and the access to the hash table in terms of computation time is constant on average.

Building Trusted Execution Path

Both learning and protection modules, are composed by a tracing execution path component that is the core of our detection system and is in charge to efficiently and correctly intercept (requirements R1 and R2) any native method call of any Java application that is running on the system. To achieve this goal, the system needs to be able to intercept any Java method invoked after a deserialization takes place in an application. In particular, we are interested in intercepting all the execution path in form of stack trace in the following form, for every deserialized class `A` and every corresponding native method call `X`:

```
A.readObject()
method1()
method12()
...
native call X
```

In order to extract the stack traces we first need to intercept the native methods and then parse the JVM stack memory structure to collect the invoked Java methods. To this end we analyzed several approaches for our design. We first considered dynamic bytecode instrumentation, but we found it unsuitable for our purpose since native methods cannot be simply instrumented, as they do not have bytecode. In order to overcome this problem we first need to create a wrapper for each target native method, and then redirect to it all the calls inside Java code that point to that native method. Although there exists a mechanism to perform this operations [49], it would work only for classes that have not already been loaded, since the JVM does not allow insertion of extra methods to a class that is already loaded. As we would need to dynamically add methods (the wrappers) to all the classes, including ones that have already been loaded when instrumentation starts, this technique does not serve our goal.

Another idea could be to instrument every possible method in every loaded class, and check every `invoke` bytecode instruction to see whether it points to a native method. This approach fails as well, because at the time of instrumentation it is not known whether the resolved method will be native or not. Finally, trivial logging of all the method calls after `readObject` via bytecode tracing/instrumentation would be too expensive in terms of performance (R2) and the system would not scale.

After these considerations, we decided to directly modify the Java Virtual Machine to accomplish this task. Our implementation for tracing Java class methods consists of a modification to the template interpreter generator. Specifically, we modify the generation of native method entries by adding a call to our custom logging functions inside the VM runtime environment. With this approach we have two advantages:

- The system does not need to know which classes are going to be loaded, nor we have to instrument each one of them. Moreover, by running inside the JVM, our component can inspect every call to native methods, on any

class (R1).

- Effectively extract only the information of our interest, focusing on native methods and their ancestor `readObject`. This gives a significant advantage in terms of performance (R2) compared to the naive solution of forward method logging starting from `readObject` calls.

We found that instrumentation of the interpreter alone was not enough to achieve the entire coverage of all the native calls. The JIT compiler constantly looks for code optimization, and our modification to the interpreter has no effect on JIT-compiled code. Since we cannot make assumptions of how much code will be compiled on the analyzed systems, and since we want to get an accurate view of all native calls, we also instrumented the generation of wrappers for native methods defined in the JIT compiler framework. By adding our custom log logic to each of these wrappers, we are able to check the stack trace every time a native call is made. In order to test the correctness of our approach, we ran Java in fully compiled mode (with option `-Xcomp`), and in fully interpreted mode (with option `-Xint`). We found that the interpreter component does not log any JIT compiled method, and vice versa.

Input stimulation

In order to stimulate the deserialization process, we perform a static analysis on the Java code, searching for classes that implement the `Serializable` interface and define the `readObject` entry point. Starting from such classes we perform a manual analysis and figure out the inputs that can stimulate object deserializations. We also collect a set of inputs from the normal use of the analyzed Java frameworks, logging every object deserialization we observe. It is important to note that input stimulation is not a contribution of our work; our system is designed to improve the whitelist mechanism in two directions: (1) providing a better detection model (precise execution path) in terms of detection so overcoming the static analysis limitations that effects actual whitelist methods, (2) and create an automatic extractor system that can operate at runtime with low overhead and it is able to reconstruct and detect a precise execution path. An automatic system for improving input stimulation is discussed in the Discussion section.

1.3.4 Experimental evaluation

In order to evaluate our approach we analyzed two real-world Java frameworks, JBoss and Jenkins, broadly used in several companies and IT infrastructures. We also chose these frameworks since there are real attack samples available for them [28]. For each application we derive a metric that is able to show the reduction of the attack surface considering our approach. The metric, Java Class Invocation Attack Surface (JCAS), compares the number of Java classes observed during the training phase in the deserialization context with the number of potentially available classes in the monitored application's classpath. In this

context, a class is *observed* in the training phase if at least one of its methods appears in at least one collected benign stack trace. Given the percentage p of the classes that were observed during monitoring, compared to all the classes in the classpath, the JCAS metric is expressed as the percentage $100\% - p$. Such a metric is able to capture the attack surface reduction since it shows how our detection model is able to restrict the set of actions of an attacker. In fact, with our detection system in place, the attacker needs to follow the execution path of the benign deserialized data and he cannot choose the gadgets (e.g. Java classes) among the entire classpath of the Java vulnerable application. Our metric exactly shows this reduction.

We also computed the overhead for each application considered in our experiments. All tests were performed using a custom build of OpenJDK 8 with our system enabled, and a clean OpenJDK 8 build as a baseline for comparison in overhead. The tests were run on a quad-core, Intel Xeon machine with 8 GB of RAM running Ubuntu 16.04 LTS.

JBoss Application Server

JBoss is an open-source Java application server, broadly used in industry. We tested JBoss 5.1.0 with our framework. We collected the benign stack traces related to native method invocations in deserializations that occur during normal operations. In particular, we stimulated the following operations: (1) server start and shutdown, (2) application deploy/undeploy, and (3) use of management consoles and deployed applications. We also trained our system on JBoss for a period of one week. During this period, several operations were stimulated by a group of users that produced hundreds of deserialization events. We collected a total of 13298 stack traces from native calls made by deserializations. We analyzed the classes necessary for computing our metric, in particular we found a total of 43250 Java classes in the JAR files present in JBoss' classpath, plus a total of 6005 present in the Java standard libraries, for a sum of roughly 49000 Java classes. The total methods called during deserialization lead to a total of 329 observed Java classes.

In Table 1.1 we can see the data and computed JCAS metric for JBoss, showing the reduction of the attack surface; we see that by applying our model, the attack surface is reduced by 99.2% considering the benign deserializations observed during the learning phase.

Jenkins

Jenkins is an open source automation server for tasks related to software building and continuous integration. It allows the creation of customizable and schedulable jobs for building artifacts and performing related operations. We tested Jenkins version 1.649. We collected the benign stack traces related to native method invocations in deserializations that occur during normal operations. In particular, we stimulated the following operations: (1) server start and shutdown, (2) job creation and customization, and (3) job scheduling and running.

Table 1.1: Attack Surface Reduction

Application	JCAS	Native method calls	Total classes in stack traces
JBoss	99.2%	13298	329
Jenkins	99.8%	6526	74

Also for Jenkins we trained our system for a period of one week, asking a group of user to access it and use it via web. As a result of our entire test, we collected a total of 6526 stack traces from native calls made by object deserializations. We analyzed the classes necessary for our metric: we found a total of 23493 classes in the JAR file present in Jenkins' classpath, plus a total of 6005 present in the Java standard libraries, for a sum of roughly 30000 classes. All the methods observed during deserialization came just from 74 classes. In Table 1.1 we can see the data and computed JCAS metric for JBoss, showing the reduction of the attack surface; we see that by applying our model the attack surface is reduced by 99.8% considering the benign deserializations observed during the learning phase

Effectiveness

For each application we tested the effectiveness of our approach by running the real-world attack payloads provided in the ysoserial repository [28]. In particular, we ran payload CommonsCollections1 that uses reflection and runtime-loading Java class mechanisms against JBoss, and validated its vulnerability leading to arbitrary code execution with our system disabled; we also ran payload CommonsCollections5 against Jenkins, with the same result. Afterwards, we applied our protection by running the applications within our defensive framework. We found that our system can effectively block such attacks on both applications, after an appropriate learning phase. Both applications were then tested over a period of one week with our protection enabled, by exposing them via web to a group of users. Normal operations (including, but not limited to the ones listed above for each program) were triggered in both applications, leading to hundreds of deserialization events. No false positives were found by our system, even when enforcing the execution of all the methods in learned stack traces for an observed deserialized class.

Overhead

In this section we analyze the overhead introduced by our system. We performed a micro-benchmark and a macro-benchmark to evaluate the sandbox efficiency. The micro-benchmark focuses on local sandbox performance; we measure the time taken by our checks on the stack traces to validate native method calls. The macro-benchmark measures the overhead introduced in whole applications from the user's perspective.

Table 1.2: Micro-benchmark results.

Sandbox component	Mean	Standard deviation
Interpreter	$2.071x10^{-5}$s	$6.640x10^{-5}$s
Compiler	$2.883x10^{-5}$s	$6.613x10^{-6}$s

Micro-benchmark For our micro-benchmark, we measured the time required for our checks made at each native call to analyze the stack trace, reconstruct the sequence of calls made from the `readObject` call onwards and compare it to the policy learned in the training phase; table 1.2 shows the results. The test has been conducted on a number of 10000 native call traces, on which average and standard deviation were calculated. We differentiated the checks for the compiler component and the interpreter component; we can see that the time required for checking is consistent for both, and relatively small. The overhead of the system is the result of the linear composition of the time taken for the checks, triggered at each native method call.

Macro-benchmark For the macro-benchmark, we computed the time for several common operations performed by end users in our test applications. Given the huge number of native calls performed during the process, and the determinism of the executed operations, this constitutes a reliable measurement for the total overhead. The values were computed both programmatically and manually (by triggering specific operations), and averaged over 10 measurements. Initially, we observed a massive overhead of over 900% for the whole startup process on JBoss. This inefficiency was due to the very high number of instrumented native calls, some of which were invoked hundreds of thousands of times in our measured runs, causing delays of several seconds with the sum of their individual analyses, as explained in the discussion on the micro-benchmark. After manually analyzing the most frequent native method calls, we established that we could exclude most of them without any security concerns, as their interaction with the OS is limited and does not constitute a threat, regardless of their input. For example, native methods `System.currentTimeMillis` and `Class.isArray` were excluded. With this tuning in place, we were able to substantially reduce the overhead introduced. Tables 1.3 and 1.4 show the result. While the overhead is still relevant in percentage, it is worth noting that our tuning of logged native calls can still be improved, and further drops of the overhead are expected; a minimum overhead would be reached if the set of analyzed native calls contained all and only the potentially dangerous invocation.

Possible improvements and current impact An additional reduction on the overhead could be made by improving the checks on native method calls. Currently, for each call we have to traverse its stack trace at least once to determine whether the call occurred during an object deserialization. Lighter checks could be designed so that the stack trace would not have to be traversed outside of deserializations, reducing the times measured in the micro-benchmark.

Table 1.3: Macro-benchmark results for JBoss

JBoss operation	Baseline JDK	Modified JDK	Overhead
Startup time	21.0 s	29.5 s	40.5%
Console login	2.0 s	2.1 s	5.0%
Flush connections in datasource pool	1.5 s	1.9 s	26.7%
Sample WAR deployment	2.9 s	3.5 s	20.7%

Table 1.4: Macro-benchmark results for Jenkins

Jenkins operation	Baseline JDK	Modified JDK	Overhead
Startup time	9.0 s	10.5 s	16.7%
Homepage loading	1.8 s	2.3 s	27.8%
Login	2.0 s	2.5 s	25.0%
Job saving after creation	2.3 s	2.6 s	13.0%

Another idea would be to completely disable the checks on native method calls outside of the context of deserializations, bringing the overhead close to 0% for most of the executed code. In our implementation, this would require a dynamic patch to the templates produced by the JIT compiler, which would have to be done at the beginning and the end of each `readObject` call. This investigation will be part of our future research.

An important point to consider is that currently, even without these additional improvements, the net increase in the measured operations in terms of seconds is not perceived by the end user, given the relatively small absolute values of delays in the context of web applications. We can conclude that our system shows good performance from an end user's perspective, who does not perceive any substantial delay experience when they have used the protected applications.

1.3.5 Related work

A solution to address deserialization attacks could be avoid deserialization on untrusted content, by signing serialized data and checking its signature upon deserialization. While this works perfectly in theory, in practice one cannot exclude the risk of signature counterfeiting if some bug is exploited on the signing side and an attacker gets access to the keys [33], nullifying the whole protection offered by this approach. Moreover in order to authenticate all data we need to set up a PKI infrastructure and usually such structures do not scale since they need a complex management setting. One possibility when trying to directly tackle the problem of deserialization of untrusted data is to consider a restriction of the attack surface by hardening the main deserialization entry point: the class `java.io.ObjectInputStream`. An approach to perform this type of hardening is the use of modified version of ObjectInputStream [54]. This can be done by extending the class and overriding its method, such as the

`resolveClass` method, to insert security checks and perform validation before deserializing data. A Look Ahead Object Input Stream (LAOIS) [54] is an input stream relying on this logic, "looking ahead" to check whether the data presents some problem before actually deserializing it. Such method is based on static analysis that failed when applied to some classes that resolve their methods or addresses at runtime. With the reflection technique a single dynamic method could call any method in the currently loaded application, resulting in a highly inaccurate call graph for the entire application. When subclassing is not an option (for example when the code to protect is owned by a third party), it is possible to use other solutions to modify the behavior of ObjectInputStream globally, such as the use of a Java Agent for dynamic instrumentation of the classes.

The closest work to ours in terms of methodology is [25]. In their paper, the authors characterize the application behavior based on the information retrieved on the stack, function name, parameters etc. However our system is different from this work in several ways: (1) first of all the attacks context is different: we operate on different programming language and data information, they mainly work on system calls invoked from C language, (2) our interception method is based on JVM internals, instead they intercept system calls at the operating system level. (3) The type of the attack is completely different: they are focusing on memory errors while we are focusing on deserialization of untrusted data attacks.

The use of sandboxes for protecting environments in which executed operations can be controlled and blocked is not new. In fact software compartmentalization has been proposed in several context and it is based on hardware and language-based techniques [15, 24]. Karger proposed that fine-grained access control could mitigate malware [32]. Process-based privilege separation using Memory Management Units (MMUs) has been applied to several different applications: OpenSSH, Chromium and in Capsicum although with substantial performance overheads and program complexity. More recently, hardware primitives such as Mondriaan [64], CHERI [62], and CODOMs [61] have extended conventional MMUs to improve compartmentalization performance and programmability. Java sandboxing develops a mature and complex policy mechanism on top of language, but leaves open the possibility of misbehaving native code. Language-based capability systems, such as Joe-E [39] and Caja [40], allow safe compartmentalization in managed languages such as Java but likewise do not extend to native code. In our work we design a sandbox system that is able to intercept native methods by modifying the JVM internals. As we already showed in the paper we have tackled several design and implementation challenges in order to make our system handle the tracing of Java applications starting from the native calls.

1.3.6 Discussion

In this section we present limitations of our approach, and possible future improvements of our system.

Data attacks

We now describe an hypothetical attack that bases its effectiveness on manipulation of the data inside the JVM's memory, as opposed to direct execution of arbitrary code. Such attacks are already present in the memory errors area [11]. Suppose that the gadget explained in section 1.2.3 was not used to reach an endpoint for instantaneous remote command execution (or other malicious effect), but was instead targeted to invoke some method or change some fields in a class (possibly via reflection), that could later be triggered, changing the normal control flow of the application and causing the malicious effect to be activated.

Listing 1.8: Data attack entry point

```
1 public class Commands {
2   private static lstCommandString = "ls -al";
3   public static lstCommand() {
4     Runtime.getRuntime().exec(lstCommandString);
5   }
6 }
```

Listing 1.8 shows a naive example of this possibility: if the class was in the classpath and if the attacker was able to modify field lstCommandString with a gadget, class Commands would be compromised and later use could lead to remote command execution. By design, the analysis performed by our system is limited to the temporal frame of `readObject` calls; moreover it focuses on native method calls to build a recognition model for attacks. The example just presented makes it clear that with our current approach we cannot block this type of attack, due to these limitations. In order to overcome the threat posed by data attacks, our system would also have to instrument the access to data and either allow or deny it based on predefined policies; an example of policy could be to deny all write operation on sensitive data during object deserializations.

Native call restrictions

Another limitation of our approach is evident when considering classes that by design eventually invoke one or more potentially dangerous native calls when deserialized. Our system, as currently designed, would learn that such calls are normal benign behavior during the learning phase (provided the class is appropriately stimulated); if the execution of such calls included user controlled data (e.g. if one of the native calls used a field of the class as input), an attacker would be able to obtain malicious behavior slipping under our radar, with relatively little effort (a gadget exploiting a vulnerability of this type could be as simple as a serialized object with a particular value for a String field). One possible solution for this would be to make our model more fine-grained in the future, making it able to take into account not only native method calls and

their stack trace, but also their parameters. An appropriate learning strategy would then need to be developed, to learn what constitutes benign input.

Improving learning

The precision of our model is limited by the coverage obtained in the application during the learning phase. If learning is performed manually, even with prolonged use by experienced users that voluntarily explore the application and trigger benign deserialization behavior it is easy to argue that some possible attack entry points could never be observed, if the application is sufficiently complex. Moreover, the learning process is hard to engineer, and we cannot offer generic guidelines on how to perform it for any application. For this reason, in the future our system could benefit from a component for automated learning. This component would analyze source code (or bytecode) in applications to automatically detect where possible entry points for deserialization are located. Via static code analysis techniques, it would be possible to try and trace the execution paths that lead to such entry points, of course after facing challenges and limitations for static Java code analysis [37]. Combined with dynamic program analysis, we could measure the coverage of the deserializations found, and 1) try to generate benign variations of input to further stimulate the program automatically; we could also 2) detect if an entry point was not stimulated at all, producing a report advising users to test it.

1.4 Towards automatic detection and validation of deserialization vulnerabilities in the Java Virtual Machine

In this project, other researchers and I applied static analysis techniques to the problem of deserialization attacks. Our aim is to explore and develop techniques for automatic generation of deserialization exploits, which can help developers validate their code by ensuring that it's exploit-free before publishing. A publication on the results we obtained is currently under writing.

1.4.1 Overview

The goal of our analysis is to find out, given a specific Java library, the relationship among its classes and their methods in terms of execution. More specifically, our goal is to discover valid serialization chains that are also exploitable. To find such chains, we first need to build a call graph that shows the relationships between methods of the analyzed classes. Afterwards, we need to extract valid deserialization chains that reach an exit point of our interest. Among such chains, we need to identify the ones which are exploitable. In order to verify such properties, we need to tackle several challenges.

Call Graph Construction

The first challenge to solve is related to the call graph generation. The first trivial solution is to look at invoke instructions in Java bytecode, and build the call graph from them. While this is a good starting point, it is not sufficient to construct correct relationships among methods. For example, if we consider the payload CommonsCollections2 in ysoserial (see Section 1.2), we can observe that there is a transition between class `PriorityQueue` and class `TransformingComparator`. However, `PriorityQueue` has a field of type `Comparator`, which is a generic interface, and not specifically of type `TransformingComparator`, which is an implementor of such interface. Such missing information (i.e., the link between `PriorityQueue` and `TransformingComparator`) can lead to an incomplete graph, and produce false negatives in our analysis. We need to consider this when building the graph, and include interface implementors and (for the same reason) class extenders. Additionally, we create a link between methods in the graph only when at least one of the following conditions is satisfied:

- (1) The method's class implements the `Serializable` interface.
- (2) The callee method's class is a superclass of the caller method's class
- (3) The method has the `static` modifier.

If neither condition holds for a given method, it will not be possible to validate the chain that contains it. It is important to note that all the objects (i.e.,

methods) that appear in the chain should be serializable. The only exceptions to such a case are calls to methods in a non-serializable superclass (condition 2), or calls to static methods by directly invoking the method from the java class (condition 3).

Exploitable Chain Validation

Once we have built the call graph, we extract the valid chains with entry and exit points of our choice. Such chains need to be validated, and in case flagged as exploitable. As seen in the previous Section, not all valid chains are exploitable. To see an example of this, we consider Listing 1.9. In the code we have method `Example.example` that concatenates two strings, method `StringBuilder.append`, which is called when performing such operation, and method `String.valueOf`, called by the second. A correct call graph must link them, and the following chain is always exploitable (with no input needed); i.e., an execution of `Example.example` always results in the execution of the entire chain:

```
C1:  Example.example -> StringBuilder.append -> String.valueOf
```

However, what happens if we want to continue the execution beyond the last method? Just by looking at the call graph, we see that `String.valueOf` calls method `toString` on its `Object` parameter, so we should be able to assign an instance of `Object` (i.e. anything we want) to the parameter, and proceed from there. While this reasoning is correct if we were considering only method `String.valueOf` (or equivalently, a chain starting from it), in this case it would certainly lead to errors, such as considering this chain as exploitable:

```
C2:  Example.example -> StringBuilder.append -> String.valueOf ->
Example.toString
```

which is incorrect, as the parameter `var0` can only be of type `String` (propagated from `Example.example`, specifically it is the string `"BAR"`). We conclude that the call graph alone has insufficient information on how to generate exploitable chains. In the example just shown, `C2` is a false positive, while we are interested in finding exploitable chains such as

```
C3:  Example.example -> StringBuilder.append -> String.valueOf ->
String.toString
```

Listing 1.9: Call graph precision example

```
1   class Example {
2     public example() {
3       return "FOO" + "BAR";
4     }
5   }
6
7   // class StringBuilder
8   public StringBuilder append(Object var1) {
9         return this.append(String.valueOf(var1));
10  }
11
12  // class String
13  public static String valueOf(Object var0) {
14        return var0 == null ? "null" : var0.toString();
15  }
16 }
```

In order to solve tackle this challenge we need to design a custom static data-flow analysis, combining reaching definitions analysis and type propagation analysis. The idea is to build an inter-method data dependency graph, containing information on control flow and data dependency between variables, also across links in the chain. Next, by propagating the variable types in the graph, we can mark type inconsistencies, i.e. situations like the example of Listing 1.9 seen above, in which specific propagated types collide with broader parameter/variable types in later calls; in the example we have the `String` type restricting the possible types of parameter `Object var0`, causing a type inconsistency in all the links in the graph between method `valueOf` and a class different from `String`. This inconsistent links can be pruned from the graph, as they always represent non-exploitable paths; in the end, if the pruned graph still contains a path from entry point to exit point, we mark the chain as exploitable, and validate the path through manual analysis. Otherwise, we mark the chain as non-exploitable.

Search algorithm

Having analyzed the requirements for finding chains in the call graph, we implement a depth-first search (DFS) to explore it. The call graph is built on a finite set of classes, which is the first input parameter of our search algorithm. Entry points and exit points are also customizable, and a maximum depth can be specified. At each step of the search, we evaluate the conditions specified in Section 1.4.1, and stop the search if our criteria are not met. If, at any point before reaching the maximum exploration depth, we find a path from entry point to exit point, we output it as a potentially exploitable chain.

Validation algorithm

After we have stated what problems we want to solve with respect to chain exploitability, we design a validation algorithm that will be able to exclude the false positives generated in the search phase (due to the causes analyzed in Section 1.4.1). Given a single chain, our goal is to establish whether it is exploitable. In order to detect if it is a false positive, we want to track data dependency among variables in the bodies of its methods. In order to accomplish this, we build an inter-method data dependency graph, in the following way:

- For each method in the chain, we generate the control-flow graph (CFG), and trace data dependency starting from a reaching definitions analysis.
- For each link in the chain `M1 -> M2`, corresponding to a call to `M2` in the body of `M1`, we match the arguments in the call statement in `M1` to the corresponding variables in `M2`

At this point, we have the necessary information to track data dependency among all variables in the chain. We can now apply a type propagation analysis, and detect whether there is some case like the one analyzed in Listing 1.9, in which some link may not exist because of type compatibility. In this situation, we are able to prune such an occurrence from the graph, and proceed. If, at the end of our analysis, there is a path from entry point to exit point, it means that it has passed our analysis, and we deem the whole chain exploitable. Viceversa, the absence of a path means that the chain is not exploitable, and it is marked as a false positive in our search.

1.4.2 System implementation

We now describe the implementation of the algorithm, after having given an overview in the previous section.

As we have seen, our strategy is made of two phases: 1) a search phase, in which valid and potentially exploitable chains are extracted from a given classpath, and 2) a validation phase, in which single chains are analyzed in detail, to eliminate false positives. We match this distinction in our implementation, and build two separate tools: **ChainsFinder** and **ChainsAnalyzer**, respectively in charge of the first and the second phase.

To implement both tools, we leveraged the capabilities of Soot[60], a framework for static analysis and transformation of Java bytecode. Soot works by generating an intermediate representation (IR) of Java bytecode[1], on top of which various analyses can be performed.

Figure 1.1 schematizes the architecture. The input classpath (consisting of the base JARs of the Java Runtime Environment (JRE), and an input JAR for the library we want to explore) is fed into ChainsFinder, the tool for building the call graph and extracting the potential chains. The identified potential chains

[1]Soot offers different intermediate representations. We use the Jimple IR for our implementation.

are passed to ChainsAnalyzer, the component responsible for performing the data-flow analysis and identifying the exploitable ones. Both components are described in detail in the next sections. Last but not least, manual analysis helps us validate the result, and construct exploits (see experiments in Section 1.4.3).

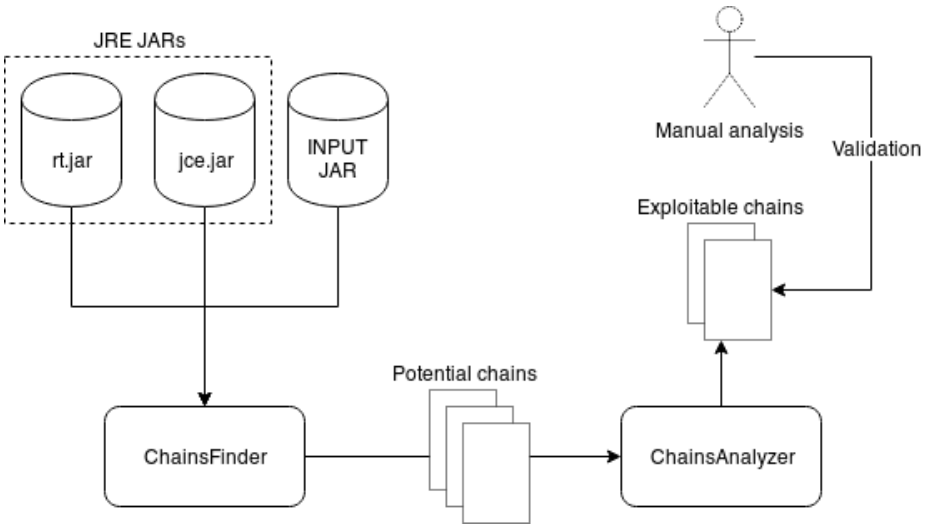

Figure 1.1: Architecture of our system for deserialization exploit search and validation

Chain search

For ChainsFinder, we leverage Soot's capability of constructing the call graph of our input classpath. Soot first generates the IR for all the classes and their methods, and then builds the call graph from Java `invoke` statements. For any invoke statement, Soot already includes class extenders and interface implementors in the possible callees, so one of our challenges is taken care of. We implement our other conditions seen in Section 1.4.1 by extracting the necessary information from the IR via Soot's API.

Intra-method data dependency

In ChainsAnalyzer, we start with the intra-method information. Soot offers a data flow framework, which we leverage to implement our analysis. The framework allows us to extend a generic data flow analysis, and specify which operations we want to perform while traversing the CFG. The CFG generation and traversal is performed by Soot, as well as the propagation of data flow sets among nodes of the CFG during the analysis.

We extend the class `ForwardFlowAnalysis`, and implement a reaching definitions analysis. While performing the analysis and propagating the data flow

information, we are able to build a data dependency graph (DDG) structure. Each node in our DDG represents a particular variable in a particular statement, and contains the following information:

- Method: determining the class and method of the current statement.
- Value: instance of `Value`, representing the current variable.
- Unit: instance of `Unit`, representing the current statement

Edges in the DDG represent dependency between nodes: there is an edge between node A and node B if A depends on B, by the following definitions:

- a use of a variable V at a node N (with value V) depends on the definition of V an the node M
- the definition of a variable V at a node N depends on the use of another variable U at a node M if N and M have the same unit
- a use of a variable V at a node N (with value different U different from V) depends on the definition of U at the node M

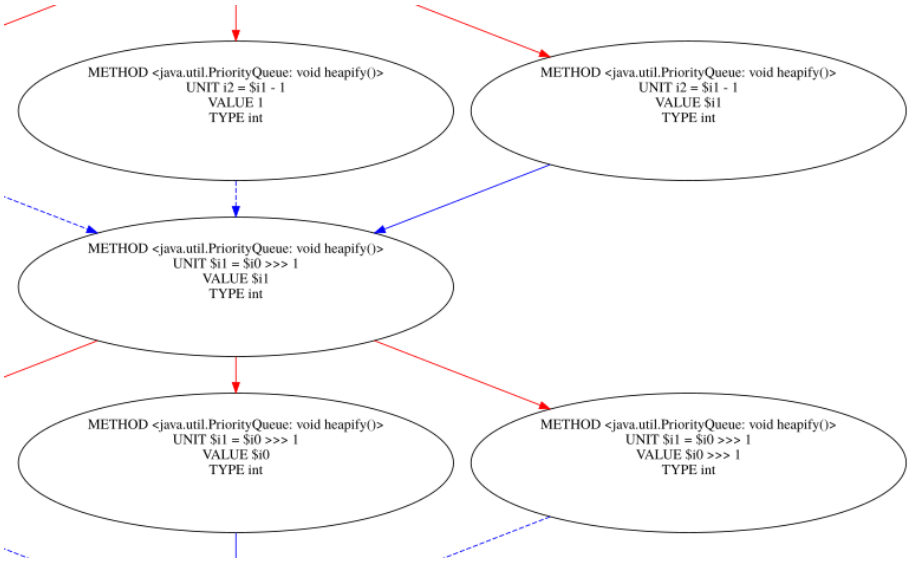

Figure 1.2: Detail of intra-method DDG for CommonsCollections2

At this point, the DDG contains only information about intra-method data dependency; Soot does not automatically handle data flow between methods, so we need to build the inter-method transitions manually. Figure 1.2 shows a section of the DDG built for chain CommonsCollections2, specifically intra-method data dependency inside `PriorityQueue.heapify`; different arc colors

match different types of data dependency between nodes, according to our definitions. We can repeat this analysais for every method in our chain (which we know in advance, since the analyzed chain is an input of our program), and add nodes to the DDG.

Inter-method data dependency

In order to validate our chains across method links, we have to add inter-method information to the graph. We can do this in our forward flow analysis by inspecting Java invoke statement, which we can match with chain links. When we find an invoke statement at method M_x in the chain, all we have to do is check whether the callee is step M_{x+1}. If so, we can create an inter-method edge in our DDG. There are two cases to distinguish:

1. inter-method parameter call - in this case, the value of node M_x is a parameter of the method call (see Figure 1.3). We track the value and make sure it is correlated with the appropriate parameter in the next method's DDG.

2. inter-method instance call - in this case, the value of node M_x is the object on which the method call is performed (see Figure 1.4). Therefore, in the CFG of M_{x+1}, such object will be referenced by the `this` pointer. Again, we make sure to create the link, and make the Jimple `@this` value in M_{x+1} depends on the value of the current node in M_x.

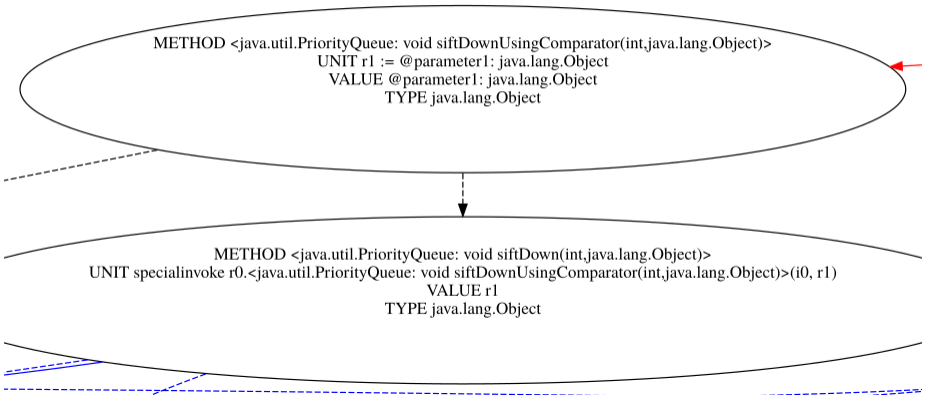

Figure 1.3: Inter-method parameter call. The value of the dependency (`r1`) is a parameter of the method call.

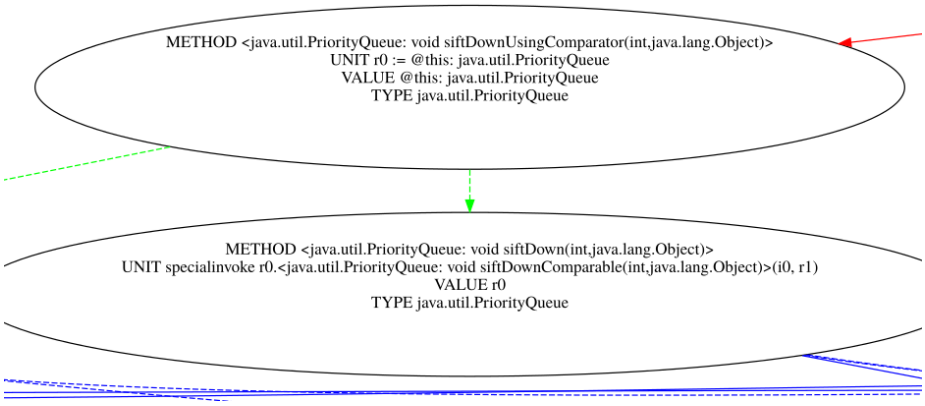

Figure 1.4: Inter-method instance call. The value of the dependency (`r0`) is the object on which the method is invoked.

Type propagation

After the creation of these inter-method edges, we are ready to perform our type analysis. We want to assign type information for nodes of which type is certain, and then propagate the information through the graph to detect any inconsistencies. We add additional information to each node:

- allowed_types: a dictionary `Map<Value, List<Type>`, containing information about the possible types for each known variable at a given node.

We start with a `null` value for allowed_types at every node (meaning the node has not been processed yet), then we initialize only nodes with no dependencies for their value: for each node N with value V and no dependencies for V, we put `V.type()` in the allowed types for V at N.

Next, we propagate type information in steps. At each steps, we process all and only the nodes which have no dependencies with allowed_types still set to `null`. In other words, we process only nodes which do not depend on nodes that are yet to process. When processing node N with value V, we copy the allowed types for each variable in its successors in its allowed_types dictionary (duplicates are removed), with the following logic: each of them is compared with the type T of V at N; only types that "can hold" T are copied and allowed for V at N (i.e., T and supertypes). If, after being processed, a node with value V has the empty set as the allowed types for V, we have found a type inconsistency, meaning that data-flow through that particular node is not possible.

Special care is taken for inter-method links, which are handled separately. The logic of type propagation is the same, but the types are matched also on the called object and the method parameters, depending on the type of inter-

method edge (as explained before).

After iterating this step of type propagation until there are no unprocessed nodes (i.e. nodes with allowed_types still set to `null`), we can prune the graph, by removing all edges of nodes with type inconsistencies.

At this point, we are left with a reduced graph, on which we can perform our final query: with a simple DFS visit, we see if there is a path from entry point to exit point, going in reverse; i.e., we search for a dependency path from the exit point to the entry point. If we find one, we mark the chain as exploitable, otherwise as non-exploitable. Figure 1.5 shows a detail of the complete inter-method data dependency graph of chain CommonsCollections2 after type analysis. The red and blue arcs describe intra-method data dependency inside `PriorityQueue.readObject`, while the green arc represents the transition to `PriorityQueue.heapify`. The type of the variable `r0` is propagated from the start of the chain to the intra-method nodes, and through the method invocation on the variable itself (its type is maintained in method `heapify`).

1.4.3 Experimental evaluation

In this section, I present the experimental results of our work. Our main aim is to build a system for improving the automatic discovery and recognition of exploitable chains. We evaluate both components of our system, ChainsFinder and ChainsAnalyzer, measuring their effectiveness.

Structure of our search

Our tools have the purpose of finding new exploitable chains in libraries, or the lack thereof, with the aim of securing them against attacks based on deserialization of untrusted data. In order to evaluate their efficacy, we must have a way of validating the chains produced by ChainsFinder and the exploitability verdicts produced by ChainsAnalyzer. As our detection is not fully automatic, and we lack formal proof of correctness for our search, we proceed by using manual analysis and known exploitable chains as ground truth, to identify false positives and false negatives and measure the accuracy of our detection.

Our ideal target result is a complete, exploitable deserialization chain that leads to arbitrary command execution. However, such chains can be very hard to find. Considering ysoserial, there are just two gadgets in all the exploits for libraries Commons Collections 3.1 and Commons Collections 4. Also, there is a performance problem to take into account, that is the time taken by ChainsFinder with respect to the maximum depth of its DFS search (which is a parameter, see Section 1.4.2). We found that the time to explore the call graph increases exponentially with the depth of the search; this fact limits our possibility of exploring the graphs with high depth parameters. However, we make an observation. If we consider an exploitable deserialization chain of length n in this form:

`Class1.readObject -> Class2.method2 -> ... -> Class`$_n$`.exitPoint`

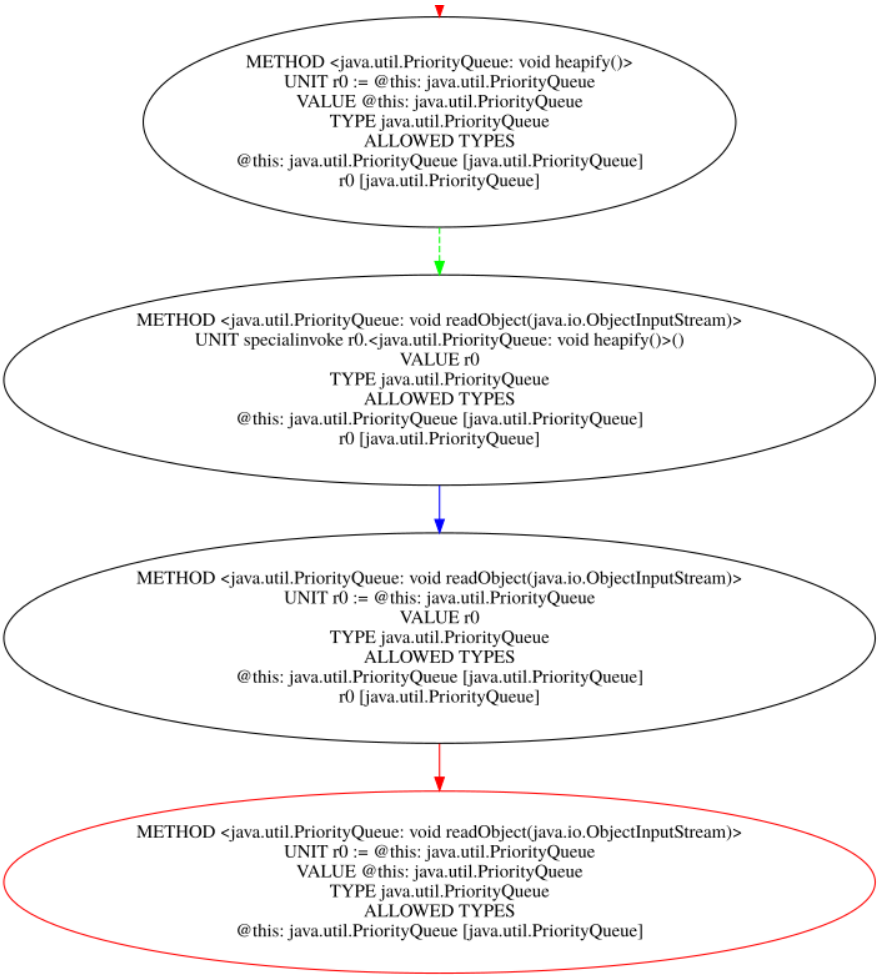

Figure 1.5: Intra- and inter-method type propagation, with allowed types list.

we see that it can be arbitrarily split in two parts, by chosing an integer $1 < m < n$:

$$\begin{aligned} \texttt{C1:}\ \ & \texttt{Class1.readObject -> Class2.method2 -> ... ->} \\ & \texttt{Class}_m\texttt{.method}_m \\ \texttt{C2:}\ \ & \texttt{Class}_m\texttt{.method}_m\ \texttt{->}\ \texttt{Class}_{m+1}\texttt{.method}_{m+1}\ \texttt{-> ... ->} \\ & \texttt{Class}_n\texttt{.exitPoint} \end{aligned}$$

We are left with C1 (which we call **trigger**), a valid exploitable deserialization chain (although its execution doesn't lead to the exit point), and C2 (which we call **gadget**), an exploitable chain (which leads to the exit point despite not being a deserialization chain). The key point to observe is that C2 is a chain by

itself: if executed, it will lead to the exit point. Knowing this, if we had a chain such as C2, we could just search for a chain like C1 to *activate* it. We would do so by searching for deserialization chains, with exit point $\texttt{Class}_m\texttt{.method}_m$. It is clear that this technique offers two advantages:

1. we no longer have to look for complete deserialization chains leading to our exit point. We can find intermediate valuable *gadgets*, and then work to find *triggers* to activate them. Due to the difficulty of finding complete chains, we can also work by finding triggers to reuse known gadgets.

2. the complexity of the search is reduced. Instead of searching chains of length n (with running time $\mathcal{O}(k^n)$ for some k), we can now search gadgets and triggers of the same length, effectively doubling our maximum chain exploration depth, or exponentially decreasing the computation time if the depth is kept constant, by performing two searches with running time $\mathcal{O}(k^{n/2})$

The approach also has a limitation: the right choice of interesting triggers is assumed for the advantages to be present. If the search of combinations of triggers and gadgets is not restricted, we still incur in an exponential complexity, because the only choice is to save every chain of our search as a trigger, and use its exit point as an entry point for potential gadgets. Of course, restricting the possible combinations means that there is a higher risks of false negatives. Only an accurate choice of interesting *intermediate methods*, i.e. intersections between triggers and gadgets, ensures that this technique effectively reduces the search complexity without cutting promising search paths. At the current state of our research, we can only rely on known gadgets for making this choice highly effective; in the future, we hope to develop heuristics and/or semi-automated approaches that will guide the search of intermediate methods.

Unit testing

For evaluating the efficacy and accuracy of our recognition, we unit tested ChainsAnalyzer on five known positive exploitable chains, and ten known negative, non exploitable potential chains found by ChainsFinder (and validated by manual analysis). These tests were built and run while developing the tools, serving as our ground truth. After finishing the complete implementation described in Section 1.4.2, the accuracy of our unit testing is 100%, with no false positives or negatives.

Results

The most relevant result produced so far by our project is the discovery of two exploitable chains on the library Commons Collections 3.1 and Commons Collections 4 [52, 46]. The payloads are not completely new: they reuse gadgets in ysoserial that link `java.lang.reflect.Method.invoke` to `java.lang.Runtime.exec` (so they are in fact triggers for such gadgets). Also, since the reused

gadgets have already been patched, the two chains do not work in newer versions of the library. The chains have been discovered with a combination of use of ChainsFinder and manual analysis. Our software was launched on library Commons Collections 3.1, with `java.lang.reflect.Method.invoke` as exit point. Since the possible entry points were searched in the scope of the library, and included any method of serializable classes (not just `readObject`), the tool found parts of the two chains, which were put together with manual analysis to produce working payloads. Although the result could certainly be improved (for instance, by finding a complete novel gadget for `Runtime.exec`), we think that it already proves the potential of our approach in finding new chains. Payload CommonsCollections8 in particular has an interesting property that differentiates it from all other previous Commons Collections payloads: its entry point (i.e. the serializable class `TreeBag`) is part of the library itself, while all other known chains have entry points in standard Java classes found in the JRE.

Besides this new payloads, so far our tool has not found any other exploitable chain (no triggers leading to `java.lang.reflect.Method.invoke` and no gadgets leading to `java.lang.Runtime.exec`). So far we have tested the following libraries, for chains of maximum depth 8:

```
apache-collections-commons-collections-3.1
commons-collections4-4.2
groovy-2.5.5
hibernate-core-5.4.1
javassist-3.24.1
jython-standalone-2.7.0
richfaces-api-3.3.4
spring-core-5.1.4
struts2-core-2.5.20
```

1.4.4 Related work

With our work, we try to automatize some parts of exploit search and construction. Automated exploit generation has various examples in literature, outside the deserialization and Java domains. For instance, Alhuzali et al. [7] construct a tool (Chainsaw) that automatically finds injection vulnerabilities in web applications; their research on PHP [8] is also relevant, where automated analysis on a codebase reveals the presence of several vulnerabilities (which are turned into working exploits).

Static analysis has already been applied in the context of bug finding and security research. In their work, Livshits and Lam [38] apply static analysis techniques to open source libraries, finding several security bugs.

In the particular context of deserialization vulnerabilities, a notable work to mention is the tool serianalyzer by Bechler [47]. Serianalyzer uses static Java bytecode analysis to trace native method calls made by methods during deserialization. Altough it produces many false positives, it has been used

to find many of the exploits present in the ysoserial repository. We decided to implement our own tool for chain discovery, ChainsFinder, to leverage the capabilities of Soot and its intermediate representation, and to extend the search of exit points beyond native method calls.

Several attempts have been made for protection against attacks based on deserialization of untrusted data. Besides the work already cited in Section 1.3.5 (which is relevant for this project as well), and our novel dynamic approach presented in Section 1.3, notable work has been done by Dietrich et al. [20]. In their publication, the authors analyze the problem of deserialization of untrusted data not only in Java, but in several affected languages. After analyzing a few chains that cause Denial Of Service, they study in detail possible mitigations for the problem.

To the best of our knowledge, our work is the first approach towards automatic static exploit validation in this area.

1.4.5 Discussion

In this section I present some limitations of our current work, and possible new lines of research on the topic.

Incomplete search scope

At the moment, ChainsFinder and ChainsAnalyzer work by finding and validating chains based on the CFG constructed by Soot. Although this has proven effective in both validating known exploitable chains, and finding new ones as well, the approach has some limitations, as it cannot detect gadgets above a certain complexity. Consider as an example the *templatesImpl* gadget from the ysoserial repository, which makes use of dynamic class loading, or the chain CommonsCollections1, which uses dynamic Java proxies. The two dynamic techniques, along with others such as reflection, are not modeled by Soot, and pose a challenge to static analysis in general. Simple modelling of dynamic techniques for class loading and method calling would most likely be imprecise and lead to an exponential increase in the call graph (due to the enormous amount of possibilities, e.g., for a reflective call with arbitrary parameters), and a lot of false positives if the search were performed without additional heuristics. We think that only other approaches, such as symbolic execution or fuzzing could have a hope of partially solving these issues.

Incomplete validation

Since our approach is based on pure static analysis, even when we find a positive we are vulnerable to two issues:

1. False positives - our approach is unable to detect whether a branch in the code will ever be taken. Trivially, Soot cannot detect a condition that is always false at runtime in a particular chain, and will therefore build edges

in the call graph of ChainsFinder even when they can never be executed. The same reasoning holds true for ChainsAnalyzer.

2. Lack of positive proof - while for negative samples we can know exactly what doesn't work for exploitability, if our sample is flagged as positive by ChainsAnalyzer we cannot automatically produce a payload to prove that it is by executing it. The search of payloads is still manual.

Future work

We now explore possible approaches that have the possibility of addressing the issues we have listed.

Fuzzing Fuzzing is the technique of feeding random, or partially random input into a program, in order to stimulate unwanted behavior and corner cases. Useful in software engineering for finding bugs, this techniques has also successfully been applied to information security [31]. In our context, we see fuzzing as a possibility for finding payloads, once a potentially exploitable chain is known. While random fuzzing has low probability of finding the correct input to exploit a chain, there is very interesting work in the world of software engineering done by Fraser et al. [26], on the use of branch distance metrics for automated test generation with high code coverage. The guidance provided by these metrics could be used to aim fuzzing in the precise direction of the wanted chain, by generating inputs that satisfy branch conditions.

Symbolic execution To solve the problem of guiding the execution through the correct conditional branches, constraint solving and symbolic execution techniques could be used. At the moment, several possibilities exist for performing symbolic execution in Java [6, 3]; however, while constraint solving works well with basic types such as integers and strings, to the best of our knowledge there is currently no modelling of custom objects in OOP. If such a model were developed, then the whole search of exploitable chains could be made more accurate, by exactly solving constraints on objects and variables, and deterministically generating inputs that allow a particular chain to be executed/exploited.

We think that the most promising road for future work is the use of fuzzing, provided that the challenge of developing appropriate metrics is overcome. Once correctly guided, we believe that fuzzing will have great potential of finding exploitable chains. The main advantage of fuzzing, compared to our approach, would be a higher level of automation, as fuzzing could be used to find possible paths for new chains *and* help to construct actual payloads for exploits; the amount of manual analysis required would decrease. As a direct consequence, the technique would scale better when increasing the size of the explored paths, although we still think that a step of validation via static analysis checks would remain useful, in order not to waste time on trying to find exploit on false positive paths. Symbolic execution could, in theory, eliminate the nondeterminism

in both our approach and fuzzing. Deserialization chains would become exact solution to a set of equations modelling the execution flow, cancelling the type-I and type-II errors introduced by heuristics, and the need to introduce randomness to find new results, as fuzzing does. However, even if the research on the topic will advance and symbolic execution will be able to model OOP Java programs, we think that two challenges would still need resolution:

- the complexity of some features could be hard to model in a symbolic execution approach. Parallel asynchronous execution, reflection, dynamic class loading, aspect oriented programming: all of these technologies are present in Java and pose a direct challenge to symbolic execution, as they greatly complicate the control flow of programs in nonlinear ways
- when translating the problem of finding exploits to the problem of finding solution to constraints and control flow equations, it could happen that some solutions would be too complex to find, in terms of the number of variable involved in the equations. In such cases, a lighter approach such as ours or a fuzzing-based one could yield better results, while still being more error prone in general.

Other research scopes

As a final remark, we think that it would be interesting to explore the possibility of transferring our research and results to other domains that include serialization/deserialization techologies. We think that our approach could easily be applied to other languages and frameworks, where deserialization vulnerabilities are already known to be present. As an example, consider Android's `Parcelable` interface, which is a substitute for the Java standard `Serializable`. While the technical details of the two mechanisms are different, the core functionality is the same:

- objects can be serialized to and deserialized from binary stream of bytes
- the behavior for serialization and deserialization can be customized

We have seen that the last point (combined with an adequate customization power, in terms of which operations are allowed during deserialization) is the seed of all deserialization vulnerabilities. The main challenge when passing from one technology to another is understanding the possible vulnerabilities that a developer might introduce when assuming that the deserialized data will be trusted. Once this analysis is complete, we think that our approach can be applied with minimal effort if the similarities with plain Java deserialization are strong enough. In the particular case of `Parcelable` objects in Android, we see no particular differences in the step of object reconstruction during deserialization: in Java we have the `readObject` method, while in Android we have the `CREATOR` field. Considered this, we think that this represents an ideal domain for our research to be reproduced.

Part II: High-level authentication on short-distance channels in Android

The following sections describe a study of authentication (or the lack thereof) in Android apps using short-distance communication channels. The work has been accepted at ACSAC 2019.

2.1 Detecting (Absent) App-to-app Authentication on Cross-device Short-distance Channels

2.1.1 Problem introduction

Cross-device communications allow nearby devices to directly communicate bypassing cellular base stations (BSs) or access points (APs) [22, 36, 35] . Such a paradigm can bring many benefits, such as spectral efficiency improvement, energy saving, and delay reduction. Without the need for infrastructure, such a technology enables mobile users (e.g., Android) to instantly share information (e.g., pictures and videos) with each other, even in areas without cellular coverage or access points [43]. It is also becoming an important technology for mobile social networks [21]: friends close to each other can be automatically identified and paired up. Moreover, this technology is used to establish the so-called mobile ad-hoc clouds, which take advantage of unused resources of nearby devices to provide cloud services, such as data and computation offloading. This is also a typical case of IoT environment, where IoT devices communicate with each other on short-distance channels [18].

Several solutions exist for securing cross-device communication. In the Android environment, they allow authentication of devices and communication channels [19, 42]. However, these solutions are not sufficient to protect the

entire communication flow. Specifically, the proposed protection system in [19] restricts apps' access to external resources, such as Bluetooth, SMS and NFC, by defining new SEAndroid types to represent the resources based upon their identities observed from their channels. The policies bind an app to a particular device on a specific channel. In this case, a malicious app installed on one device, which is allowed to communicate with a paired phone, can interfere with the communication and inject data on the channel. This problem is due to the fact that the authentication between apps is missing, and such authentication is needed in addition to the device-level and channel-level authentication. One can solve this problem by designing Android access control at system level for preventing an unauthorized access to communication channel (e.g, Bluetooth) during security operations, and removing public resources for stopping side-channel attacks [42]. This, however, makes the system less usable and compatible for the apps that already use the public resources for legitimate purposes. Moreover, these systems do not handle channels such as: SMS, Audio, Wi-Fi and NFC. We name this security issue cross-device app-to-app communication hijacking, or *CATCH*. We argue that *CATCH* is critical and is due to the fact that no APIs or mechanisms are made available to Android programmers for performing app-level authentication on short-distance channels (e.g., Bluetooth, Wi-Fi-Direct).

In our work, we study the problem of mutual authentication between two apps running on two different devices and communicating over a short-distance channel. Although such channels already provide device pairing and authentication methods, these methods only operate at the device or channel level. They are oblivious to the apps running on the devices. In this study, we first define the authentication scheme for short-distance channels. We then design a new tool that is able to analyze a given Android app and detect potential *CATCH* vulnerabilities (i.e., the lack of app-to-app authentication). Our tool uses several data-flow analysis techniques and is able to recognize specific *if-statement* conditions in the code related to the authentication scheme. Such particular conditions can be precisely recognized since, in our context, the analyzed authentication model must be performed with some sort of dynamically generated secret (out-of- band authentication) (Section 2.1.3) that is usually stored in the dynamic memory (e.g. heap, stack). We perform some experiments to show the flexibility of our tool on detecting authentication schemes, even when the target app has been manipulated with the ProGuard obfuscator, one on the most used obfuscators for Android [4]. Our tool can be deployed in several contexts: it can serve as a tool for the developer, or it can scan apps in distributing environments (e.g. Google Play) for detecting potential vulnerabilities on Android apps that communicate by using short-distance channels.

In summary, we make the following contributions:

- We identify a **security problem** called cross-device app-to-app communication hijacking (CATCH), which commonly exists in Android apps that use short-distance channels, and afflicts all the tested Android versions. We perform experiments on a dataset that contains 662 Android apps that

use Bluetooth technology, collected in the Androzoo repository.

- We provide a **solution** to the CATCH problem by designing and developing an authentication scheme detector that analyzes Android apps to discover potential vulnerabilities. We tackle several challenges in identifying code boundaries of the authentication scheme, along with the authentication checks.

- We **validate** the results of our system on Android apps with manual analysis, and test its resilience in detecting the authentication scheme. The results show that our approach produces 0% of false positives and false negatives. We also show two case studies on real Android apps.

The problem we study fits in our broader research goals, stated in the introduction of this thesis. As per the Java deserialization works presented earlier, we are again dealing with a high-level, multifactorial security issue, which involves protocol architecture, the design of specific features of the operating system, and coding practices of single developers. While it is hard to identify a "silver bullet" solution to patch the problem at any level, we are able to tackle the problem with the following steps:

1. We identify and model the problem's environment

2. While we can't cover 100% of possible authentication protocols that may be affected by *CATCH*, we define a threat model which we repute both broad and realistic enough to capture the essence of the problem, and perform meaningful study regarding its treatment

3. We leverage control flow and data flow analysis to treat the problem in the scope of out threat model; specifically, we build a system that allows us to analyze Android APKs and detect functional high-level features (in our case, authentication over a channel) based on the constructed flow graphs.

Although our analyses alone are not sufficient to completely overcome the problem, they allow us to analyze its presence in a dataset of real-world Android APKs, and build tools that can be deployed to raise the awareness of developers' towards the problem, and/or to help secure app markets by performing static analyses on uploaded APKs.

2.1.2 Background

In this section we provide the necessary background to understand the security vulnerabilities in Android apps performing peer-to-peer communication.

Authentication for Cross-device App-to-app Communication

We study the problem of mutual authentication between two apps running on different devices and communicating over a short-distance channel. Although

such channels already provide device pairing and authentication methods, these methods only operate at the channel level: they allow two devices to be paired and mutually authenticated (i.e., establishing a channel) but they are oblivious of the apps running on the devices (i.e., all apps on these devices share this established channel). As a result, when two devices are paired and authenticated at the channel level, it is possible for a malicious app on one device to interfere in a communication on the channel between two legitimate apps.

Currently, most cross-device, peer-to-peer communications channels are authenticated by using an *out-of-band* scheme that works as follows. A user (**requesting user**) A initiates a communication from his device to a nearby device, whose user (**accepting user**) B is then prompted with confirmation. The confirmation is requested either with a secret PIN that B has to communicate to A via a separate channel (e.g., verbally), or as a simple "accept" button presented along with information that enables the identification of the device trying to initiate the communication. These steps are already implemented in Android; one never needs to re-implement authentication for the communication channel. Once authentication has passed, communication can begin. Bluetooth uses encryption to protect the channel. It is important to note two points related to authentication in this scenario:

1. Authentication occurs via sharing of *out-of-band* information/secret.
2. Authentication performed on the channel (Figure 2.6) is not sufficient to guarantee authentication between higher level applications communicating over the channel.

Point 1) is important as a general property of authentications performed in our scenario. The exchanged information needed to confirm authentication is, in practice, visual and verbal contact between the two users, and the out-of-band element is a constant in all this type of authentications. More strongly, we exclude the possibility of authentication being carried out exclusively via information passed on the same channel being authenticated, as a result of previous research [13, 56].

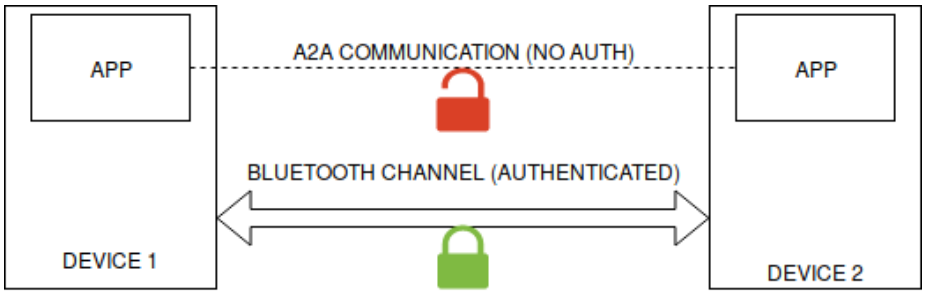

Figure 2.6: Authentication of A2A communication is not guaranteed by channel authentication

To understand why the lack of app-level authentication is dangerous, let us consider the following example (Figure 2.7): a chat app using Wi-Fi-Direct opens a `ServerSocket`, accepting communication through it and display incoming messages to the final user.

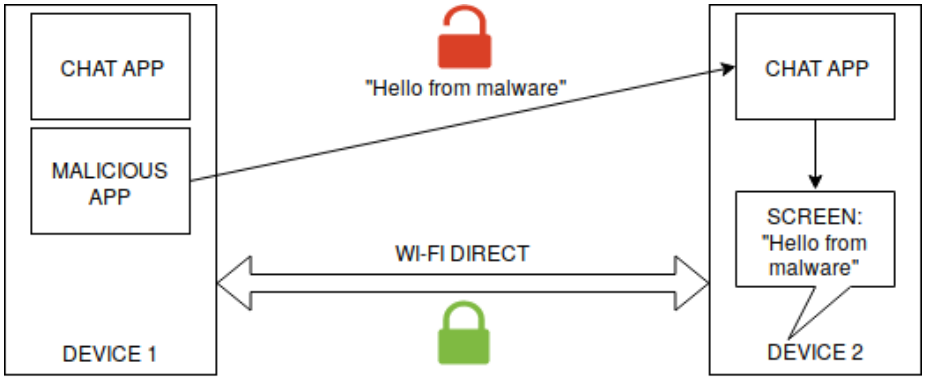

Figure 2.7: Malicious app sending content to chat app

The intended use of the app is to be installed on two devices that communicate with each other in a peer-to-peer fashion. We also consider the presence of a malicious app on one device, this is a common threat model, as shown in [42]. Since the *devices* are authenticated, and not the *apps*, the malicious app has permission to communicate over the channel, as any other app installed on the device. The malicious app can therefore craft custom messages to send to the other device, which are displayed as if they were sent from the original app. If there is no code performing authentication in the benign app, there is no possibility of detecting this sort of action.

Depending on the particular context, there are some scenarios in which the attack can become very dangerous:

- *Phishing*: in cases like the example described above, the malicious app could send phishing material to the other app. The user will be likely to trust and open the content, as he will have no means to distinguish it from benign content sent from the device communicating with him.
- *Malware delivery*: the same system could be used to deliver malware to the user, in the form of malicious files that would trigger a vulnerability upon opening (for instance, a malicious PDF file that targets a vulnerable PDF reader).
- *Exploitation*: if the target app performs internal operations depending on commands received from the communication channel, the malicious app could send commands that could change the execution flow and trigger unwanted behavior. For instance, a command to delete the user data could be issued to a file manager app that accepts operations via Bluetooth device.

It is important to note that other network attacks such as MiTM are difficult to accomplish in this context, since the attacker should be physically close to the devices in order to hijack the communication, and would also need to overcome or bypass the channel protections, such as encryption. For this reason we believe that the attack explained above is the most realistic in this environment. We name the underlying problem common to all these scenarios **cross-device app-to-app communication hijacking**, or *CATCH*. Having established the potential impact of the problem, we aim at building a system for automatic analysis of Android apps, targeted at detecting the presence (or lack thereof) of authentication on particular communication channels. The purpose of our system is to provide a tool that will help app developers to secure their software. Moreover, our detector can also be used as a security scanner on app markets (e.g., Google Play) for detecting potential authentication vulnerabilities.

2.1.3 Approach overview

This section describes the design and structure of our approach. We build our system with the goal of automatically verifying the existence of app-to-app authentication in Android apps. To detect app authentication in an automated way, we mainly face the following challenges:

C1) We need to define a generic scheme that captures the essential logic of app-to-app authentication. Such a scheme is necessary for identifying and evaluating the implementation of authentication in apps. (Section 2.1.3)

C2) We need to define a strategy for differentiating between an if-statement that does not operate on security critical data and an if-statement that is a part of the authentication scheme. (Section 2.1.3)

C3) Additionally, the authentication scheme can be implemented in several ways according to the developer experience. This adds an additional layer of difficulty for our analysis, that should be general enough to also capture such cases. (Section 2.1.3)

We now proceed to illustrate our approach for building an analysis tool that is able to tackle these challenges and provide accurate results in terms of detection.

Authentication Definition

In this section we define an authentication scheme for cross-device communication in Android environment. More specifically our authentication model considers two devices, D1 and D2, with apps A1 and A2 respectively installed. The two devices establish an authenticated channel, on top of which A1 and A2 initiate a communication. Such form of authentication proposes authenticated information exchange between mobile devices using several methods different than the standard RF channel [56]. These are called out-of-band, side-channels or location-limited channels (LLCs) [58], and include audio, visual, infrared,

ultrasound, and other forms of transmission [45, 57, 53, 12]. Such techniques allow the receiver to physically verify the source of the transmission. Using this information, the devices are mutually authenticated, and a secure shared key can be established. More precisely in such an authentication scheme we recognize the following steps:

1. A2 obtains a secret that will be used to authenticate communication. This secret is either generated on device D2, and then communicated to app A1, or it is generated by D1 and then shared with A2. Such a communication uses an *out-of-band* channel which is also called a "human assisted channel". Such a channel cannot be manipulated by an attacker, and thus it is considered trusted by definition.

2. Once A1 and A2 share the same secret, they can start sending data, using the secret as authenticator. Depending on what the secret is and the application protocol, the data could be encrypted with a key derived from the secret (e.g., *Hash(Secret)*), or the secret could be sent as plaintext along with the data for authenticating the transmission.

3. In both cases (encryption with key derived from the secret, or secret sent with data as a simple pass-phrase/PIN), app A2 needs to perform authentication checks on the received data. In the first case, A2 needs to check that the decryption operation performed by the secret key is correct, and in the second case A2 needs to check whether the pass-phrase/PIN is correct. These checks must occur before any critical use of the data, otherwise the communication is not authenticated. Only in case the checks are correct, the data is authenticated and the communication can continue.

We mentioned "authentication checks" that are performed in step 3. It is crucial to define what form these controls might assume, in a way that helps us target their recognition in code. Moreover such a definition should be general enough to capture the majority of several forms of the authentication schemes deployed by different developers. We define a **communication** in our model as some exchange of data from A1 to A2, beginning when A2 reads the data from the communication channel. We define a **use** of the data as any operation whose result depends on the data itself. We define an **authenticated use** of the data as any instruction that needs to be authenticated before access to the data. We give the following **definition of authentication** in our model: given a communication over a peer-to-peer channel with exchanged data `D`, an authentication is a condition in code situated between the beginning of the communication and the first authenticated use of `D`, which either: (1) allows the execution to continue, in case `D` is successfully authenticated, or (2) prevents any authenticated uses of `D` every time the authentication is unsuccessful. The internal logic of the authentication checks depends on the context, and is therefore not possible to include it in the definition.

Listing 2.10: Sample Bluetooth socket communication

```
try {
  socket = mmServerSocket.accept();
} catch (IOException e) {
  Log.e(TAG, "Socket's accept() method failed", e);
  break;
}

if (socket != null) {
  InputStream inputStream;
  try {
    inputStream = socket.getInputStream();
    byte[] buffer = new byte[10];
    inputStream.read(buffer);
    if (buffer[1] == 10) {
    writeToFile(buffer);
    FunctionLibrary fl = new FunctionLibrary();
    writeToFile(fl.return6());
  }
  mmServerSocket.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
  break;
}
```

Detection of Authentication Scheme

For detecting authentication, we first explore the possibility of identifying authentication schemes via the use of particular APIs. If such APIs existed, then we could reduce our analysis to a code reachability problem. This is the case, for instance, of authentication over Unix domain sockets [55]. Unfortunately, we could not find any standard APIs for app-level authentication for the technologies we analyzed. For this reason, we shift our focus on detecting a set of instructions in the code that might indicate the presence of an authentication mechanism. In such a context we must clearly define a strategy for identifying possible authentications once we track the data of our interest. The first step for creating a scalable analysis framework is to identify boundary code points in the application. Such boundary permits to restrict the analysis only to a part of code that potentially could contain an authentication scheme. After the boundary area is identified we can apply further code analysis techniques in order to validate the authentication scheme. In our system the boundary area is defined by two main elements: the *entry* and *exit* points.

More specifically, an *entry point* is an instruction in the code that indicates

the start of the communication over the analyzed channel (e.g., data receiving). Given this broad definition, we can recognize multiple entry points in an application for a given communication. For example, In Listing 2.10 we can see an example of Bluetooth communication in Android app. Since the data is read from the stream at line 13, the instruction represents an entry point. The call `socket.getInputStream()` at line 11 is also an entry point for this communication. We are obviously interested in entry points that help to indicate the start of communication for a specific channel such as Bluetooth. An accurate identification of the entry points for a communication channel will ensure that all possible communications over such channel are identified and targeted by our analysis.

The end of the boundary is defined by an *exit point*. An *exit point* is represented by the first *authenticated use* of the data coming from the monitored channel. Even though exit points exist for every communication, it is hard to define whether an exit point is an authenticated use of the data or not, since this is a semantic property of an use. As an example, the use of line 15 in Listing 2.10, where the data is written to file, may or may not be an authenticated use, depending on what the file is used for. If it is a log file used simply for debugging purposes, and virtually never checked unless an error occurs, then it is not important that authentication necessarily occurs before such point. On the other hand, if the data defined into the file is part of the main flow of the app protocol, then authentication must necessarily occur in order to avoid untrusted and potentially dangerous data in the file.

Due to this ambiguity of the use of the data, we design a detection strategy that is not dependent on exit points. In particular we design an algorithm (Algorithm 2.1.1), based on program analysis techniques, that performs data and control flow analysis. The algorithm starts computing the Control Flow Graph (CFG) and Data Dependency Graph (DDG) for each analyzed app (line 7-8). Both graphs are necessary to find out the relationships between data of our interest and the condition statements that depend on such data. Then, for each node in the CFG, the system determines whether it is an entry point by using function `isEP`. This function uses a pre-defined table based on function signatures related to a specific communication channel (Section 2.1.4). If no entry points are found, the result `NO AUTH NEEDED` is returned (lines 9-12). In all the other cases, each node in the DDG is analyzed. If the node represents a condition in the code (function `isCondition`), then the system checks if there exists a path in the DDG that connects an entry point to the conditional node (lines 16-17).

Algorithm 2.1.1: Authentication detection

```
1  input: APK app
2  output: NO AUTH NEEDED |
3          NO AUTH FOUND |
4          POSSIBLE AUTH FOUND
5
6    entry_points ← []
7    cfg ← computeCFG(app)
```

```
8     ddg ← computeDDG(app)
9     foreach node in cfg
10      if isEP(node) then entry_points.add(node)
11    end
12    if entry_points == [] then return NO AUTH NEEDED
13
14    foreach node in ddg
15      if isCondition(node) then
16        foreach ep in entry_points
17          path ← findPath(ep, node, ddg)
18          if path != null
19          then
20            if isCheckConstant(node, ddg) == false
21            then return POSSIBLE AUTH FOUND
22          endif
23        end
24      endif
25    end
26
27    return NO AUTH FOUND
```

If such a path exists, it means that we possibly found an authentication scheme. However it is still possible to obtain false positives: simple sanity checks or other controls on data would be all erroneously identified as authentication. In order to reduce the number of false positives among conditions that are candidate for authentication, the algorithm applies a *constant propagation* technique. Technically speaking, such technique is using reaching definition analysis results. In particular, if a constant value is assigned to a variable, and such variable is not modified before a point `P` in code, then the variable has a constant value at `P` and can be replaced with the constant.

In our context, since the analyzed authentication model must be performed with some sort of dynamically generated secret (out-of-band authentication, Section 2.1.3) that is usually stored in the dynamic memory (e.g., heap, stack), by using constant propagation we can discard all the conditions that use constant values in their comparison, as they certainly do not represent authentication on data. Constant propagation is a very powerful technique for our analysis, and it helps to reduce the false positives to 0% in our experiments as we will show in Section 2.1.5.

2.1.4 Implementation details

We now discuss our practical implementation choices for the algorithm presented in Section 2.1.3, by describing the technical details of our system.

Overview

We implemented our system on top of the Argus-SAF framework [63]. The framework offers various tools for analyzing Android apps, such as the generation of the CFG and DDG that we need in our algorithm. Also, the framework translates Dalvik bytecode into an intermediate representation (IR), called Jawa, on which our algorithm performs the analysis. In particular, various conditions in code, including `while` and `for` loops, `if` statements and exception

`try/catch` blocks, are all translated into `if` statements in the intermediate representation. The CFG and DDG built by the framework contain nodes that map to single Jawa instructions, making it possible to have the fine-grained, instruction-level information that we need in our algorithm for targeting conditions. Furthermore, Argus-SAF permits inter-component modeling, meaning that transitions between components such as Android intents are integrated in the graphs. These features made possible for us to explore the application code together with the graphs built by Argus-SAF on top of it. Our system is composed of three main components: (1) Graphs Builder, (2) Path Finder and (3) App-to-app Authentication Finder. Our framework accepts an app in input (as an APK file), and outputs either that no authentication has been found, or a list of specific instructions in code that may contain authentication checks.

- The *Graphs Builder* starts the Argus-SAF analysis on the APK. The framework applies four sequential steps: (1) the Jawa IR is generated from the Dalvik bytecode, then (2) an environment model of the Android system is generated. This is crucial to capture the control flow and interactions between components, such as the dispatch of intents between activities. (3) At this point, Argus-SAF builds an inter-component control flow graph (ICFG) of the whole app. At the same time, it performs data flow analysis and builds an inter-component data flow graph (IDFG) on top of the ICFG. (4) Finally, the framework builds a data dependency graph (DDG) on top the IDFG. We mainly use information from this graph in our analysis. The information of our interest is extracted in the Graph Builder by using classes `ComponentBasedAnalysis` and `InterComponentAnalysis` for extracting the CFG and DDG. The graphs are then passed to the next component.

- The main goal of the second component, *Path Finder*, is to locate areas in the code where an authentication scheme may exist. This is done by identifying data flows for the protocol of our interest, and performing reaching definitions analysis to see if any conditional statement operates on data read from the channel that we are inspecting. The Path Finder component traverses the CFG received from Graphs Builder, and marks entry points for the analyzed channel based on a predefined list of method signatures. It then finds all conditional statements, which is accomplished by extracting all the nodes of type `IfStatement` in Argus-SAF. At this point, it is possible to perform reaching definition analysis, to check whether there is at least one conditional statement using a variable that was earlier defined as data read from the channel. The DDG obtained from Graphs Builder contains all the information to perform this search: definition-use pairs map to edges in the graph, so Path Finder traverses it in order to find possible authentication paths. It sends the discovered paths, if any, to the last component.

- *App-to-app Authentication Finder* applies further checks to the paths received from Path Finder, in order to exclude false positive results by rec-

ognizing checks against constant values. In particular, it analyzes the `if` statements in the Jawa IR, which can be divided into two types: (1) comparisons between two variables, (2) comparisons between a variable and a constant. The system immediately discards the conditions of the second type from our search, as they certainly do not represent the authentication scheme that we look for (see Section 2.1.3). For conditions of the first type, our system uses constant propagation to determine if one of the two variables in the condition is a constant. It walks up the DDG from the `IfStatement` to their definition, reconstructing the value-history of the variables from their initialization. If the last-assigned value to either of the two variables (before the `IfStatement`) is a constant, then we are in the same case of type-two conditions, and the path is again discarded for the same reasons.

Choice of Entry Points

In our implementation we focused on Bluetooth, since it is the most used technology in Android apps for short-distance communication. Wi-Fi-Direct is still not very common among the Android apps, in fact in the dataset that we analyzed we only found a few samples (10) of it. To show the security issue of CATCH applied on Wi-Fi-Direct channel we analyzed one of these apps as a case study (Section 2.1.7). However, the core of our analysis is orthogonal to any communication channel. The only part that can change among different channels is the identification of the entry points. For Bluetooth communication based on `BluetoothSocket`, we found two possible entry points (i.e., where a `BluetoothSocket` stream starts receiving data): `BluetoothSocket.getInputStream` and `InputStream.read`. A typical Bluetooth communication flow involves the former function, called to obtain an `InputStream` object, followed by an invocation to the latter function. In the DDG of an application containing this type of communication, the instructions operating on data read from the channel are linked to both functions. It would appear that `InputStream.read` is the best choice for an entry point: semantically, it actually represents the point in which the data from the stream enters the control flow of the app. However, given the general use of class `InputStream` outside the context of Bluetooth communication, this choice led to many false positives in practical experiments. For this reason, the choice of `BluetoothSocket.getInputStream` worked much better as definition for our entry point for Bluetooth. Although it is an instruction *preceding* the actual read operation of data from the Bluetooth stream, it uniquely identifies our protocol of interest. Moreover in all the communication flows that we observed in Bluetooth apps operating on BluetoothSockets, the functions are always used in pairs.

2.1.5 Experimental evaluation

In this section we present and discuss the results about the experiments we performed to validate our system.

Preliminary considerations

In order to test the efficacy of our algorithm we need to collect a balanced dataset that contains both positive (i.e. apps with authentication at application level) and negative samples (i.e. apps without authentication). In our analysis we noticed that the security problem of CATCH afflicts all the Android apps using Bluetooth in our dataset. For this reason the dataset is unbalanced. To test our system under such conditions, we divided the experiments into two main categories: (1) a dataset analysis on APKs retrieved from a research repository [9], aiming at confirming the efficacy of the algorithm on negative samples; (2) a targeted analysis on custom apps built by applying code transformation techniques (e.g. obfuscation) for proving that the authentication scheme is correctly detected by our algorithm.

Dataset analysis of Android apps

To evaluate the efficacy of our system, we ran tests on a large number of APKs collected from the Androzoo repository [9]. The Androzoo dataset contains more than three million unique Android apps, crawled from several Android markets: Google Play, Anzhi and AppChina. In our experiments we pre-filtered APKs from the dataset and selected non-obfuscated apps that use Bluetooth. We decided to focus only on Bluetooth apps considering the amount of manual analysis we performed during the design of our algorithm, which could help us as a ground truth for validating our results. We started analyzing a total of 210,425 APKs, randomly chosen from the Androzoo repository. In order to select the appropriate Bluetooth APKs we applied the following filter: check if an app (1) requires the Bluetooth permissions in the manifest file; (2) contains certain libraries and classes related to Bluetooth (e.g., `BluetoothSocket`). The filter produced a total of 2,739 APKs.

We then applied a second filter where we exclude the obfuscated apps since it is quite hard validate them at this first step. For this filtering we focus on the ProGuard obfuscation tool, which is the free software most commonly used by developers, and it is referred in the Android Documentation [4]. In particular, we implemented some heuristics for recognition based on the typical class names (e.g. `a.class`) produced by ProGuard in obfuscated APKs. This filter selected a total of 942 APKs from the initial set of 2,793, which means that the majority of the apps in our dataset, almost 70%, use ProGuard for code obfuscation. After running our algorithm, we discovered that 704 of the selected apps do not have any entry point for Bluetooth communication in the CFG. This happens in cases where Bluetooth functionality is imported in some library/classes, but never used in the code, so the instructions that we would mark as entry points for our analysis never appear in the CFG/DDG. We also excluded such APKs from our dataset. Finally, we obtained a number of 238 APKs, suitable for our analysis and evaluation.

We then performed our first experiment. We ran our system on the 238 APKs without constant propagation enabled. This experiment shows the important

role of the constant propagation technique on reducing false positives. It shows that 26 APKs out of 238 are found positive (i.e., about 11% of the APKs are potentially performing authentication on data read from Bluetooth sockets) and the rest are found negative (i.e., not performing authentication). they never applied any checks on data received form the Bluetooth channel.

In our second experiment, we enabled the constant propagation technique and we ran our system on the same set of 238 APKs. In this case we observed that all of the positive samples found previously were actually false positives (i.e., they used one constant value in the parameter of the if statement marked as possible authentication). This result shows that no app in the dataset performs app-to-app authentication when using Bluetooth.

At this point, we manually investigated the negative cases to check for any false negatives. To this end, for validating our results we chose a sample of 20 random APKs from our dataset of 238 APKs and we manually analyzed them. We observed that all of them receive data from the Bluetooth channel, but they never apply any checks on such data before using it. Our manual analysis found 0 false negatives. Our experiments show that 100% of the analyzed APKs in our dataset which perform Bluetooth communication using Bluetooth sockets are potentially vulnerable to the CATCH attack model.

Dataset composition

We analyzed the composition of our dataset to make sure that we did not run tests on sample/unused/abandoned apps. We sampled 300 APKs (containing permissions/classes for Bluetooth) from our dataset, and performed a manual analysis by searching them on Google Play. We found that about 30% of the apps were present on this market. We classified the apps by category, depending on their description. The vast majority of apps belongs in the following categories:

- *Game apps*, where Bluetooth is used for playing peer-to-peer
- *IoT apps* for specific devices, where Bluetooth is used to send and receive data from the controlled device or sensors
- *Business apps*, using Bluetooth to send data from smartphone to computer, or again smartphone to device

Other categories with less APKs included health apps, used for communicating with medical devices, cryptocurrency-related apps, and smart home management apps.

Targeted analysis

For our second analysis, we built a custom app using Bluetooth. It only performs these basic operations: it reads from a `BluetoothSocket` when the user triggers an action, and it displays any received content on screen. We then patched the app to include a basic authentication scheme fitting our model: upon starting,

the app generates a random secret PIN of 4 digits, and shows it on the screen. This secret needs to be communicated out-of-band to other apps interacting with ours (e.g., verbally to another user wanting to send data). When reading from the `BluetoothSocket`, the app first expects to receive the PIN in plaintext, in the first four bytes read from the socket's `InputStream`. If the PIN matches the one generated by the app, the communication is accepted; otherwise it is rejected and the user is informed of the event. We found that our algorithm correctly predicts the possible presence of authentication.

We ran another test to check if changes introduced by common optimization and obfuscation tools would impact our algorithm. In this case we validate the obfuscation transformation since we can check the ground-truth provided by our application. In particular, we used the ProGuard tool [2] on our sample app, since it is the most commonly used by developers and it is recommended in the Android documentation [4]. ProGuard performs a series of transformations aiming to remove unnecessary code, and renames types and variables to hinder reverse engineering. We ran ProGuard on both versions of our test app (with and without authentication). We found that the transformations introduced by this tool do not impact the detection capabilities of our algorithm, which correctly discriminates the apps' behavior. In particular, we observed the following results:

- Sample app without authentication and ProGuard disabled, the system returns `NO AUTH FOUND`.
- Sample app without authentication and ProGuard enabled, the system returns `NO AUTH FOUND`.
- Sample app with authentication and ProGuard disabled, the system returns `POSSIBLE AUTH FOUND`.
- Sample app with authentication and ProGuard enabled, the system returns `POSSIBLE AUTH FOUND`.

Analysis of obfuscated APKs

Our results from the targeted tests indicate that ProGuard transformations do not affect the precision of our tool in the detection of authentication. For this reason, we decided to run our tool on ProGuard-obfuscated APKs from our dataset. We selected the 1797 obfuscated APKs that were initially discarded, and filtered them for Bluetooth use and appearance of entry points in the CFG/DDG as we did for non-obfuscated ones. This process yielded a total of 424 APKs, which we analyzed (combined with the previous experiments, we have a total of 662 APKs analyzed that use Bluetooth technology). 100% of the APKs were identified as negative (i.e., not containing authentication) by our tool, with constant propagation enabled. To validate this result, we manually analyzed 15 APKs, randomly chosen from the obfuscated APKs dataset. Since our tool indicates where the entry points are located in the CFG, and what

the possible authentication paths have been analyzed, we were able to manually validate the absence of authentication checks, confirming that our heuristic approach is not only powerful enough for detection, but also that it is resilient to the obfuscation techniques.

2.1.6 Performance Analysis

In this section we report the time needed for each phases of our analysis.

Time Threshold One of the main critical point for our analysis is how to set a time threshold for building the CFG and DDG in Argus-SAF, since the computational complexity explodes for large applications, and the system is not able to construct the entire graphs within reasonable time. After this threshold is hit while analyzing a single component in an APK, Argus-SAF will stop its analysis and move to the next component. In order to set up a correct time threshold we need to be sure that the constructed CFG and DDG include the Bluetooth entry points and the authentication checks (if present). To this end, we performed some experiments on APKs collected in our dataset. In particular, for each analyzed app we first built the graphs by setting a certain time threshold `T`, and we then search for Bluetooth entry points inside the computed CFG. Afterwards, we compute the number of nodes that are dominated by the entry point node in the graph that represents the number of instructions that can potentially include the authentication scheme. We start with a threshold of `T` = 30 sec., and then increase the value to `T` = 60 sec. and `T` = 120 sec. By comparing the different results, we notice two important things: (1) for any entry point, both the number of reachable nodes in the CFG and the number of data dependency nodes in the DDG are sufficient to contain a potential authentication scheme. More in details we found on average more than 10,000 instructions that are dominated by the entry point and the CFG reachability from a single entry point to any node in the graph is always above 99%, an expected result given by the inter-component connections in Android code. (2) The variation of the results between the three runs is minimal, that it means that we generally do not miss any important information that would have been considered adding more time of analysis. For this reason we chose a threshold of 30sec. for our experiments.

Time of Analysis For our tool a use case would be code validation where the detector could serve as a pre-release tool to check for unauthenticated communication. In such a context the tool should perform its analysis in a short-time. In this direction we perform several experiments that show the overhead of the analysis. In particular the experiments were performed on a laptop running Ubuntu Linux 17.10, with a Intel Core i7-6700HQ CPU (2.60GHz) and 16 GB of RAM. We specifically measured the time taken to analyze the 26 apps that were found positive by the first version of our system (without constant propagation). The average time spent for modeling the APK in Argus-SAF is

5 minutes, while the average running time of our algorithm on the generated graphs is 2 minutes, giving a total average time of 7 minutes. Although the variance is high, we think that even the worst-case execution time is suitable for the use cases we designed, considering that the release of an app is not an instantaneous process, and that an average of 10 minutes is a feasible testing time for an automated developing pipeline of Android apps. Moreover we can decrease the time threshold for building graphs from 2 minutes to 30 sec. and gain more efficiency by reducing the average time from 7 to 5 minutes in total.

2.1.7 Case studies

In this section, we present two real attacks case studies that we select from our dataset in which our analyzer gave negative results. Such applications are representative of the common type of applications that can be used in peer-to-peer communication environment: (1) chat app, (2) data sharing app. We will now discuss the attack implementation, and the engineering effort required for its setup and execution along with its own limitations.

Data injection on BluetoothChat

We target the Android BluetoothChat app [1]. This app is a working example of peer-to-peer chat that is affected by CATCH problem, since it does not implement any app-level authentication scheme. The BluetoothChat app gives the user the possibility to scan for nearby devices, connect to one of them by using RFCOMM identifier, and then send text messages via Bluetooth. In this attack scenario we will describe a *data injection* attack to a remote device.

Attack Preparation To accomplish a successful attack we need to satisfy two preliminaries requirements: (1) the malicious app needs to recognize the presence (i.e., installation) of the target application on the device. (2) the malicious app needs to detect when the target application is opened and run on the device. These two states, installed and opened, allow the malicious app to identify a potential active connection between BluetoothChat applications on different devices.

In order to detect the presence of the target app, the malicious app can retrieve a list of installed apps by querying the `PackageManager` object. Such operation is not privileged and it can be executed by any app installed on the device. For the Bluetooth Chat sample, the malicious app can detect the installation of it just looking at the package name. Once the presence of the vulnerable app has been identified, the next step for the malicious app is to exploit a legitimate communication for spoofing content and deliver the attack payload. However, this may happen at unpredictable time since the malicious app does not know when a remote communication will be activated. While it is possible for the malicious app to continuously try to exploit the communication by using polling technique, this is not desirable from the attacker's perspective since it creates suspicious events that can be detected. The best result would be

achieved if the malicious app could monitor the vulnerable app, and perform the attack only at the appropriate time. While it is very difficult to fully monitor the behavior of other apps from another app [65], a possible way to partially achieve the result is to monitor the list of open apps, obtainable via the `ActivityManager` class, specifically with the `getRunningAppProcesses` method. Again, this information can be requested to the Android system by any app without any specific privilege; the malicious app can continuously poll this list, and try the attack when the communication is open and running in foreground.

Payload Running If the attacker has satisfied the previous two requirements the attack can be performed successfully. In particular for the BluetoothChat case, the attacker needs to install a malicious app on one of the two devices that performs the communication. The communication protocol over Bluetooth is implemented with BluetoothSocket, with a RFCOMM identifier for the chat service. If the attacker knows the identifier, his malicious app can send messages to the app on the other device, which will be indistinguishable from benign ones. In this case, the app is open source, so the RFCOMM identifier is embedded inside the application, and a simple manual investigation can reveal it. Once the attacker knows the identifier he can perform data injection on the remote device, and send a message to a remote application. The impact of this data injection is potentially high especially if the receiver trusts the sender and for instance she is opening any forwarded links, which in this case could lead to phishing pages controlled by the attacker. The following figure shows an example of hijacked communication in BluetoothChat. The first two messages are written by the user on the device Huawei P9 Lite, while the third is sent by our malicious app; the receiving user on device Nexus 6 will be unable to distinguish the malicious messages.

Data injection on Wi-Fi Direct +

In this second case study we focus on real-world app that use Wi-Fi Direct + [5] collected from the Google Play market. This file sharing app has more than 500,000 downloads, it is constantly being updated, and it has a paid version, Wi-Fi Direct + Pro. This information definitely indicates that the app is relevant for our analysis. Since this apps does not implement and app-to-app authentication as revealed by our tool we can perform a data injection attack.

App Functionalities Wi-Fi Direct + offers the possibility to share file between two Android devices, via Wi-Fi-Direct protocol. After performing Wi-Fi-Direct pairing, on one device the user should select the option for receiving files. At this point, his device is entering in listening mode for incoming connections. When the user on the other device selects the option for sending a file. A success dialog is then displayed on the receiving device and the file is transferred.

Payload running After pairing has been established, the app on the receiving device opens a `ServerSocket`, and accepts connections on it. If a malicious app on the sending device tries to connect to the socket, we have a typical CATCH scenario, in which the receiving app is not able to distinguish legitimate and malicious data. For the attack to succeed, some technical details have to be considered. The attacker needs to study the Wi-Fi Direct + protocol used to send files in order to replicate it without errors, then he has to build a malicious app for sending files with this protocol. At this point, by activating the sending of a file from the malicious app at the right time, as explained in the previous case study, the attacker is able to inject data in the communication with another device. With Wi-Fi Direct + in particular, the useful time window for data injection is reduced in comparison to the BluetoothChat case study; this is because Wi-Fi Direct + on the receiving device will accept only one file before closing the communication channel, as opposed to BluetoothChat, which keeps listening for incoming messages. This is a problem for the attacker: if the benign app sends its file first, then the file sent by the malicious app will not be accepted (race condition). If the opposite happens instead, the benign file will be rejected, and the sending user will be notified of an error. Depending on the situation, the users might verbally communicate and establish that something suspicious happened. This risk is always present, but it is greatly reduced in cases such as BluetoothChat, where no error messages are displayed to the users. Although we recognize this problem for the attacker in some cases, we have to also consider the situations in which the users are not able to identify the attack. For instance, the sending user receiving an error message may think of a bug in the app, especially if the receiving user confirms that the file has been correctly received (his app will display the correctly received malicious file). In our experiments we were able to perform the attack successfully. We were able to run the malicious app and send a malicious file without causing any alarm on the target device.

2.1.8 Related work

To the best of our knowledge, we are the first to explore the potential dangers associated with the lack of app-to-app authentication in Android apps. However, previous research has made important contributions in related areas.

Security of Android communication channels Previous work highlighted the problems existing in Android device-to-device communications [42, 19]. In particular, Demetriou et al. [19] studied several channels of communication in Android (such as Bluetooth, SMS, Internet and audio) showing that the security model of Android does not offer adequate measures for protecting certain secrets. To address this problem, they build a security system, called SEACAT, to enforce fine-grained protection on the above resources. Our work continues the exploration of missing security features in Android, and how apps can be vulnerable if developers make the wrong assumptions about the security of the underlying system.

Security models of peer-to-peer protocols Claycomb and Shin formally studied the problem of authentication in mobile devices [13], and use BAN logic to prove that device authentication using a single communication channel is not possible. We consider this result when building our model: in particular, this justifies our assumption of the secret exchange happening out-of-band. Shen et al. focus on Wi-Fi Direct technology, studying its security and discussing related best practices [56]. Again, importance of out-of-band channels to obtain authentication is highlighted, and used in the implementation of a secure Wi-Fi Direct protocol.

Using static analysis for detecting authentication Static analysis techniques have been extensively used in previous work, for instance for detecting malicious application logic on Android or Web application[27, 44, 14], and for detecting privacy leaks in both iOS [23] and Android [29] apps. Closely related to ours is the work of Shao et al., which studies the presence of authentication in the use of Unix domain sockets on Android [55]. We followed the same choice of tools used to perform the analyses, favoring Argus-SAF [63] (formerly *Amandroid*) over FlowDroid [10] because of its superior handling of inter-component communication. An important difference is that we could not model our problem with as a standard taint-analysis reachability search, so we had to build our custom data dependency analysis on top of the tools provided by Argus-SAF.

2.1.9 Discussion

In this section we discuss about limitations of our analysis along with impact of the problem that we found out.

Impact of the problem

From our research, it is clear that high-level, app-to-app authentication is almost never present in Android apps that communicate on channels such as Bluetooth. Aside from the results of our algorithm on the large dataset, we could not manually find apps performing this type of authentication for the specific channels of our interest. We postulate that this is due to a lack of awareness in programmers, who build their code relying on sources such as the official Android documentation, and the network Stack Overflow for learning how to use a particular technology (e.g. Bluetooth). It is common to reuse sample code snippets from these sources with minimal adapting [41]; since they seem not to address the problem we are stating in any way, the issue is propagated, and any app using these technology is potentially vulnerable. It is worth noting that the actual impact of the vulnerabilities, as well as the difficulty of hypothetical attacks, greatly depend on the functionalities of the specific app being attacked, and need to be evaluated on a case-by-case basis. The evaluation of the general danger introduced by the lack of app-to-app authentication on a large scale is out of the scope of this research. We think that generally, vulnerabilities based

on apps accepting unauthenticated content would be medium-impact (as in permitting phishing and/or DOS at best), but we cannot exclude the existence of particular apps where it would be possible to obtain more severe effects (e.g. arbitrary code execution).

Limitations of our analysis

From the experiments, our system shows excellent performance in detecting the presence of CATCH vulnerabilities in Android apps. However, the results have to be considered together with the limitations of our technique. Our analysis suffers from the general limitations of static analysis. One of these limitation concerns the precision of the model of apps control flow. Argus-SAF is not able to handle particular intra-component and inter-component transitions, such as ones performed with reflection, and it cannot correctly model concurrency [63]. In practice, reflection is not commonly used by Android developers to perform normal tasks such as transitions between Activities. Instead concurrency is definitely present in peer to peer apps; to avoid blocking input/output, separate threads are typically spawned on demand to handle read/write operations on the channel (this applies to both Bluetooth and Wi-Fi Direct). In case of authentication, we expect to see controls on data read from the channel immediately after a read operation, following our authentication model. So, while it is true that it would be a problem to correctly model authentication flows that involved concurrent operations, there is no reason to expect authentication occurs in a separate thread in reality (see Listing 2.12). To further validate our results (Section 2.1.5) we manually analyzed 20 Android apps from 662 dataset apps and we check whether threads functions defined in the apps include any authentication scheme (false negative results). Our manual analysis shows that no one of the app functions threads analyzed contained any authentication scheme.

Listing 2.12: Threads in Bluetooth communication

```
// Main thread code
new ReadThread().start()

...

// ReadThread code
public void run() {
...
  if (socket != null) {
  InputStream inputStream;
  try {
    inputStream = socket.getInputStream();
    inputStream.read(buffer);
    // We expect authentication happening here,
    // not in a separate thread
}
```

Chapter 3

Conclusion

In my thesis, I applied the use of static and dynamic analyses to the topic of software protection in the Java environment. We have seen how control-flow information can be useful in all the problem that have been faced:

- in protecting the JVM, dynamic control-flow information has been crucial to reconstruct the execution trace, enabling distinction between benign and malicious samples
- in deserialization chain analysis, an accurate CFG has been the basis for derivation of the call graph and the data dependency graph
- in the analysis of Android APKs, an accurate model of the app's control-flow has been necessary to perform our data dependency analysis

We have also seen the benefits brought by data-flow information:

- in exploit validation, data-flow analysis and type propagation have possible to distinguish between exploitable and non exploitable deserialization chains, effectively improving the precision of a naive call graph search
- in Android app analysis, type propagation applied to the inter-method DDG has enabled us to detect the absence of authentication schemes in code (based on our threat model).

Even though they are affected by the discussed limitations, the positive results obtained in the presented novel works have proven the effectiveness of the applied techniques, in the limited research scope of each work.

Some of the limitations are fundamentally tied to the type of the performed analyses, for instance in the case of reconstructing information related to reflective method calls with static analysis (a problem which affects both chain generation/validation in Java and authentication analysis in Android). By applying multiple techniques to the same problem, it is sometimes possible to build a comprehensive defense approach. In our work on the topic of Java deserialization attacks, we have tried to view the problem under different perspective;

we have designed both a dynamic defense system, which can be applied to the JVM without changing application code, and we have also made steps towards automatic exploit generation and validation with our tools ChainsFinder and ChainsAnalyzer.

Some security problems in high-level software are due to the architecture of the framework that developers have to work with, and also their lack of awereness of security issues. This statement certainly applies to both deserialization of untrusted data (in which the developer of a class is rarely aware of its potential interaction with other benign code which could lead to security problems) and high-level app authentication in the Android environment, which not only lacks any sort of API to allow developer to perform such important task in a standardized, secure way, but also has no documentation on the problem. In our work, we have tried to show such risks, and highlighted the importance of either building frameworks that are secure by design, to prevent unintentional bug introduction from the developers, or raising developer awareness on potential security issues, possibly providing them with semi-automated software tools for code validation.

I personally think that the situation regarding the problems we studied is going to evolve, and in a positive direction. Awareness of information security topics is constantly growing in both the software engineering community and the industry management, and it is in the interest of maintainers of programming languages to provide secure technologies to companies, and end users in general. At the time of writing, Oracle has already included a section regarding deserialization in the Secure Coding Guidelines for Java SE[1], stating that "Deserialization of untrusted data is inherently dangerous and should be avoided". It is important that such directions are clearly imparted and not underestimated, as education of the developers' community is crucial for software quality and security. As most areas of information security, software protection is constantly improving, as new techniques emerge for discovering and treating vulnerabilities. With software codebases becoming bigger and more complex everyday, automated and semi-automated approaches are already paramount in software validation for security. For example, the development and release cycle of Javascript engines includes vulnerability testing via fuzzing. This trend of increasing importance of such techniques gives a clear direction to research, which will keep shifting from manual to automated analysis.

[1] `https://www.oracle.com/technetwork/java/seccodeguide-139067.html` (visited-December 2019)

Bibliography

[1] Android bluetoothchat sample. `https://github.com/googlesamples/android-BluetoothChat`. Accessed: 2019-01-18.

[2] Proguard. `https://www.guardsquare.com/en/products/proguard`. Accessed: 2018-11-19.

[3] Java Pathfinder. `https://github.com/javapathfinder`.

[4] Shrink your code and resources. `https://developer.android.com/studio/build/shrink-code`. Accessed: 2018-11-19.

[5] Wifi direct +. `https://play.google.com/store/apps/details?id=com.netcompss_gh.wifidirect`. Accessed: 2019-02-15.

[6] Java Symbolic Execution. `https://docs.angr.io/advanced-topics/java_support`, 2019.

[7] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 641–652. ACM, 2016.

[8] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 377–392, 2018.

[9] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[10] 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. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[11] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 14, 2005.

[12] W. R. Claycomb and D. Shin. Secure device pairing using audio. In *43rd Annual 2009 International Carnahan Conference on Security Technology*, pages 77–84, Oct 2009.

[13] William R. Claycomb and Dongwan Shin. Extending formal analysis of mobile device authentication. *J. Internet Serv. Inf. Secur.*, 1:86–102, 2011.

[14] Andrea Continella, Michele Carminati, Mario Polino, Andrea Lanzi, Stefano Zanero, and Federico Maggi. Prometheus: Analyzing webinject-based information stealers. *Journal of Computer Security*, 25:1–21, 02 2017.

[15] Stefano Cristalli, Mattia Pagnozzi, Mariano Graziano, Andrea Lanzi, and Davide Balzarotti. Micro-Virtualization Memory Tracing to Detect and Prevent Spraying Attacks. In *Proceedings of the 25rd USENIX Security Symposium (USENIX Security)*.

[16] Stefano Cristalli, Edoardo Vignati, Danilo Bruschi, and Andrea Lanzi. Trusted execution path for protecting java applications against deserialization of untrusted data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 445–464. Springer, 2018.

[17] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code reuse attacks in php: automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 42–53. ACM, 2014.

[18] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Proceedings of 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[19] Soteris Demetriou, Xiao-yong Zhou, Muhammad Naveed, Yeonjoon Lee, Kan Yuan, XiaoFeng Wang, and Carl A Gunter. What's in your dongle and bank account? mandatory and discretionary protection of android external resources. In *NDSS*, 2015.

[20] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. Evil pickles: Dos attacks based on object-graph engineering. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[21] Mianxiong DONG, Takashi Kimata, Komei Sugiura, and Koji ZETTSU. Quality-of-experience (qoe) in emerging mobile social networks. *IEICE Transactions on Information and Systems*, E97.D:2606–2612, 10 2014.

[22] K. Doppler, M. Rinne, C. Wijting, C. B. Ribeiro, and K. Hugl. Device-to-device communication as an underlay to lte-advanced networks. *IEEE Communications Magazine*, 47(12):42–49, Dec 2009.

[23] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*, pages 177–183, 2011.

[24] Aristide Fattori, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. Hypervisor-based malware protection with accessminer. *Computers & Security*, April 2015.

[25] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 62–75. IEEE, 2003.

[26] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[27] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 377–396. IEEE, 2016.

[28] Chris Frohoff. ysoserial repository. `https://github.com/frohoff/ysoserial`, 2015.

[29] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*, pages 291–307. Springer, 2012.

[30] Volker Simonis Gotz Lindenmeier. Hotspot internals: Explore and debug the vm at the os level. JavaOne Conference, 2013.

[31] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 445–458, 2012.

[32] Paul A Karger. Limiting the damage potential of discretionary trojan horses. In *Security and Privacy, 1987 IEEE Symposium on*, pages 32–32. IEEE, 1987.

[33] Doowon Kim, Bum Jun Kwon, and Tudor Dumitras. Certified malware: Measuring breaches of trust in the windows code-signing pki. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, volume 14, 2017.

[34] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. Challenges for static analysis of java reflection: literature review and empirical study. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017.

[35] Y. Li, S. Su, and S. Chen. Social-aware resource allocation for device-to-device communications underlaying cellular networks. *IEEE Wireless Communications Letters*, 4(3):293–296, June 2015.

[36] Jiajia Liu, Yuichi Kawamoto, Hiroki Nishiyama, Nei Kato, and Naoto Kadowaki. Device-to-device communications achieve efficient load balancing in lte-advanced networks. *Wireless Communications, IEEE*, 21, 04 2014.

[37] V 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] V 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.

[39] Adrian Mettler, David Wagner, and Tyler Close. Joe-e: A security-oriented subset of java. In *NDSS*, volume 10, pages 357–374, 2010.

[40] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized javascript. *Google, Inc., Tech. Rep*, 2008.

[41] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *2015 IEEE Symposium on Security and Privacy*, pages 692–708, May 2015.

[42] Muhammad Naveed, Xiao-yong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A Gunter. Inside job: Understanding and mitigating the threat of external device mis-binding on android. In *NDSS*, 2014.

[43] Phond Phunchongharn, Ekram Hossain, and Dong In Kim. Resource allocation for device-to-device communications underlaying lte-advanced networks. *IEEE Wireless Communications*, 20(4):91–100, 2013.

[44] Andrea Possemato, Andrea Lanzi, Simon Pak Ho Chung, Wenke Lee, and Yanick Fratantonio. Clickshield: Are you hiding something? towards eradicating clickjacking on android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1120–1136, New York, NY, USA, 2018. ACM.

[45] M. K. Reiter, J. M. McCune, and A. Perrig. Seeing-is-believing: Using camera phones for human-verifiable authentication. In *2005 IEEE Symposium on Security and Privacy (S&P'05)(SP)*, volume 00, pages 110–124, 05 2005.

[46] Lorenzo Nava. Commonscollections8. https://github.com/frohoff/ysoserial/pull/116, 2019.

[47] Moritz Bechler. Serianalyzer. https://github.com/mbechler/serianalyzer, 2017.

[48] Oracle Corporation. Hotspot runtime overview. http://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html, 2017.

[49] Oracle Corporation. Interface instrumentation. https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html#setNativeMethodPrefix-java.lang.instrument.ClassFileTransformer-java.lang.String-, 2017.

[50] Oracle Corporation. Java object serialization. https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/, 2017.

[51] Oracle Corporation. The serializable interface. https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serial-arch.html#a4539, 2017.

[52] Stefano Cristalli, Hany Ragab, Edoardo Vignati. CommonsCollections7. https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections7.java, 2019.

[53] N. Saxena, J. . Ekberg, K. Kostiainen, and N. Asokan. Secure device pairing based on a visual channel. In *2006 IEEE Symposium on Security and Privacy (S P'06)*, pages 6 pp.–313, May 2006.

[54] Robert C Seacord. Combating java deserialization vulnerabilities with look-ahead object input streams (laois). 2017.

[55] Yuru Shao, Jason Ott, Yunhan Jack Jia, Zhiyun Qian, and Z. Morley Mao. The misuse of android unix domain sockets and security implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 80–91, New York, NY, USA, 2016. ACM.

[56] W. Shen, B. Yin, X. Cao, L. X. Cai, and Y. Cheng. Secure device-to-device communications over wifi direct. *IEEE Network*, 30(5):4–9, September 2016.

[57] Dongwan Shin et al. Using a two dimensional colorized barcode solution for authentication in pervasive computing. In *Pervasive Services, 2006 ACS/IEEE International Conference on*, pages 173–180. IEEE, 2006.

[58] Dirk Balfanz Smetters, Dirk Balfanz, D. K. Smetters, Paul Stewart, and H. Chi Wong. Talking to strangers: Authentication in ad-hoc wireless networks. 2002.

[59] David Svoboda. Exploiting java deserialization for fun and profit. 2016.

[60] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[61] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. Codoms: Protecting software with code-centric memory domains. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 469–480. IEEE Press, 2014.

[62] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 20–37. IEEE, 2015.

[63] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Trans. Priv. Secur.*, 21(3):14:1–14:32, April 2018.

[64] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for linux using mondriaan memory protection. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 31–44. ACM, 2005.

[65] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiao yong Zhou, and XiaoFeng Wang. Leave me alone: App-level protection against runtime information gathering on android. In *IEEE Symposium on Security and Privacy*, pages 915–930. IEEE Computer Society, 2015.