Preventing Serialization Vulnerabilities through Transient Field Detection

Pierre Graux, Jean-François Lalande, Valérie Viet Triem Tong, Pierre Wilke

Preventing Serialization Vulnerabilities through Transient Field Detection

Pierre Graux
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
Rennes, France
pierre.graux@inria.fr

Jean-François Lalande
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
Rennes, France
jean-francois.lalande@centralesupelec.fr

Valérie Viet Triem Tong
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
Rennes, France
valerie.viettriemtong@centralesupelec.fr

Pierre Wilke
CentraleSupélec, Inria, Univ Rennes, CNRS, IRISA
Rennes, France
pierre.wilke@centralesupelec.fr

ABSTRACT

Verifying Android applications' source code is essential to ensure users' security. Due to its complex architecture, Android has specific attack surfaces which the community has to investigate in order to discover new vulnerabilities and prevent as much as possible malicious exploitations. Communication mechanisms are one of the Android components that should be carefully checked and analyzed to avoid data leakage or code injections. Android software components can communicate together using serialization processes. Developers need thereby to indicate manually the `transient` keyword whenever an object field should not be part of the serialization. In particular, field values encoding memory addresses can leave severe vulnerabilities inside applications if they are not explicitly declared transient.

In this study, we propose a novel methodology for automatically detecting, at compilation time, all missing `transient` keywords directly from Android applications' source code. Our method is based on taint analysis and its implementation provides developers with a useful tool which they might use to improve their code bases. Furthermore, we evaluate our method on a cryptography library as well as on the Telegram application for *real* world validation. Our approach is able to retrieve previously found vulnerabilities, and, in addition, we find non-exploitable flows hidden within Telegram's code base.

CCS CONCEPTS

• **Security and privacy → Software security engineering**;

KEYWORDS

Android application, code verification, static analysis

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3442033

ACM Reference Format:
Pierre Graux, Jean-François Lalande, Valérie Viet Triem Tong, and Pierre Wilke. 2021. Preventing Serialization Vulnerabilities through Transient Field Detection. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21),March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3412841.3442033

1 INTRODUCTION

The tremendous number of Android users has brought malicious activities to target specifically Android mobile devices [7]. Most used applications such as WhatsApp, Facebook Messenger, Telegram or Signal, are continuously scanned in order to find exploitable security flows. Companies are ready to spend a million dollars for the exclusivity of a full vulnerability in these applications[1]. Thus, it is crucial for Android application developers to be able to secure their own codes before releasing them.

Practically, Android applications are made of Java and of C/C++ sources. While numerous efforts focus on finding insecure flows in these programming languages, Android-specific features bring new challenges to the code verification area. One characteristic, that is prominent in Android, is the segmentation of applications into smaller components that interact together. This allows the use of external components from other applications as if they were libraries, *e.g.* web page viewer. Consequently, such a feature increases the attack surface and complexifies the work of code analyzers. Additionally, a vulnerability located in a popular component may compromise various applications.

Thereby, the multi-component architecture can be prone to errors for developers. In 2015, Peles *et al.* [8] have described how a forgotten `transient` keyword in Java source code can lead to code execution vulnerability. This keyword is used during the communication process, named serialization, to tell that a specific field should not be serialized. When a Java field is set to a memory address by a C/C++ code, it should be declared transient to avoid serializing local addresses. If the `transient` keyword is forgotten, an attacker can send an arbitrary address to the application. The application may use this address, potentially leading to arbitrary code execution.

In this article, we propose a novel method to automatically detect fields that should be declared as transient. Our method uses a combined taint analysis on both the Java and C/C++ codes to track

[1] https://zerodium.com/program.html

memory addresses that may lead to a serialization vulnerability. When a memory address is assigned to a field that is not declared transient, a warning is raised in order to help developers in their tasks. Our method, offers the developers a way to check their code sources before releasing their applications publicly, protecting them from forgotten `transient` keywords. In order to help developers, an implementation of this method has been released[2].

The rest of the article is organized as follows. Section 2 recalls some specificities of Android application source code and how they leave an opportunity for serialization vulnerabilities. Section 3 formally defines the Java fields that this article is searching for in Android applications. Section 4 describes the method proposed for finding these fields. Section 5 evaluates this method against real world applications and vulnerabilities. Finally, we draw our conclusions in Section 6 and pave the road towards further analyses.

2 SERIALIZATION VULNERABILITIES

2.1 Android environment

The source code of Android applications was originally written using the Java language. Since 2019, it has been enhanced with the Kotlin language and Google encourages developers to switch to this new language. Both languages are compiled into the same bytecode named Dalvik. Thus, for analyzing the code of applications, it is safer to work on the bytecode being independent from the source code than checking the two parts of the compilation chain. In the rest of this article, we will use the term "Java code" even if we always work on the bytecode.

For performance purposes, Android allows applications to embed assembly code named *native code*. Developers use the Java Native Interface (JNI, a C/C++ library) to produce this native code. For instance, this interface allows the native code to call bytecode methods, allocate objects, and get or set fields. Each bytecode entity (class, object, field, method) has a corresponding ID that is used to interact with this entity. For example, Listing 1 shows how to use the JNI to set the value 42 to a field of the calling object. Lines 2 and 3 are used to retrieve the IDs of the class `"ClassName"` and the field `"FieldName"`. At Line 4, the field of the object given at Line 1 is set: Android ensures that native methods are called with the ID of the calling object `thisObj` (or class for static methods) and a pointer to the JNI environment `env`. The duality between the bytecode and native code complexifies the work of code analyzers since they have to handle two languages [13] and to carefully model their interfaces [9, 11].

2.2 Android component communication

Additionally, Android applications are composed of multiple components that are running concurrently. Components communicate together by sending intents. This messaging mechanism is also used to communicate between different applications. It is noteworthy that the called component is in charge of checking that the caller is legitimate before processing an input. Indeed, if the component performed privileged or sensitive operations without verifying that the caller is authorized to perform these tasks, it would lead to component hijacking vulnerabilities [2, 6, 14].

[2]TATA: Taint Analysis for Transient Analysis, https://gitlab.inria.fr/cidre-public/tata/tata-release

At the implementation level, an intent is composed of bytecode objects that are serialized. Serialized data is deserialized by the called component. The deserialization process is known to be error prone: for example, in 2014, Horn [4] has discovered a vulnerability in the Android deserialization process which leads to privilege escalation. This vulnerability is due to a missing check that deserialized objects are instances of classes that are declared serializable. To exploit this programming error, the vulnerability uses native code to obtain an arbitrary code execution.

To adjust the serialization process, developers can declare fields as `transient`. A transient field is a field that is not part of the persistent state of an object, and thus, should not be serialized. Transient fields can, for example, be used to accelerate the serialization process by not transmitting fields that can be recomputed using other fields. Additionally, the `transient` keyword is used to avoid serializing fields that have a meaning only in the current process state. For example, a field storing a memory address should not be sent to another process because the memory layout is random for each process. That is why, each object reference (object instance inside another object) should be declared transient since its value is the address of the referred object. To avoid such a time-consuming and error-prone task, the serialization process is able to handle object references automatically and reconstructs the references in the destination process. However, if a `long` (or an `int`) field is set to an address by some native code, the serialization process cannot determine whether it refers to a memory address or not. Then, it is processed as a number value and sent to the other process. If this value is used without any verification, this can lead to vulnerabilities such as crashes or arbitrary code exploitations, as shown by Peles & Hay [8].

2.3 Related work

Approaches that tries to detect information leaks are often implemented using taint propagation. This also applies for Android ecosystem for which numerous taint analyzers have been proposed. As such, TaintDroid [3] is able to propagate taint tags during the execution of the analyzed application. This kind of analysis has been ported to all aspects of Android ecosystem [10, 15]. In particular, NDroid [12] is able to propagate the TaintDroid taints to the native code. Other dynamic approaches do not use taint analysis. For example, Yang *et al.* [14] have created IntentFuzzer, a tool that sends forged intents to the tested application. After sending an intent, IntentFuzzer checks that the application does not perform any privileged action, that is the application does not leak any capability.

Nevertheless, dynamic analyses are known to be incomplete: only the executed paths are analyzed. While this is enough for enforcing a security policy at runtime, this kind of approach is insufficient for checking that a source code is free of vulnerabilities. Our proposed method targets application developers who want to check their whole code base and that is why we focus on a static approach. In this case, we want to be sure that no vulnerability remains in the application.

Static approaches have also been widely studied for solving Android security problems. In particular, efforts on detecting Android

```
1 JNIEXPORT void JNICALL field_setter(JNIEnv* env, jobject thisObj) {
2    jclass cid = env->FindClass("ClassName");
3    jfieldID fid = env->GetFieldID(cid,"FieldName","L");
4    env->SetLongField(thisObj,fid,42);
5 }
```

Listing 1: Example of JNI usage

component communication vulnerabilities have already been conducted. This can be realized by statically inspecting the application. For example, Chin *et al.* [2] have created ComDroid, a tool that statically detects if a component publicly exposes privileged operations. To this end, ComDroid checks the privileges and the components that applications declare. Lu *et al.* [6] have also worked on component hijacking by creating a tool named CHEX. This tool realises a dataflow analysis of the application bytecode. This analysis checks that user data (arguments of the component caller) are not used as input for privileged or local APIs (*e.g.* SQL query), and that data obtained by such APIs are not forwarded publicly (*e.g.* GPS location). Finally, taint analysis have also been used in a static context. In particular, FlowDroid [1] is a tool that conducts static taint analysis on Java source code of Android applications. This article is based on this tool and enhances its analysis with native code management.

On the precise problem of vulnerabilities due to a missing `transient` keyword, only Peles & Hay have brought up results [8]. They report a first attempt to detect non-transient fields that are used in native code as pointers. To find such fields, they launch an application and inspect all loaded classes for reporting all fields that could be serialized *i.e.* all non-transient fields declared in a serializable class. Results are manually checked and filtered. The primary goal of their article was not to find vulnerabilities but to show how they could be exploited. That is why they do not intend to exhaustively check the presence of forgotten `transient` fields. Since they log every serializable field, their method generates numerous false positives and is neither practical nor reliable for developers who are not security experts and who cannot investigate vulnerable flows.

Contrary to Peles & Hay, the proposed method is fully automated and detects every vulnerable field while keeping a low false positive number. Additionally, the proposed method is able to detect non-exploitable fields, that is fields which should be declared transient but do not offer any exploitation possibility. These fields still have to be reported because future code modifications could give opportunities for attackers to eventually exploit them. Lastly, our solution is usable by non-security-expert developers who nonetheless wish to secure as much as possible their application.

3 MISSING TRANSIENT KEYWORD DETECTION

Formalization. In order to determine precisely the fields that are problematic and may lead to vulnerabilities, we represent Java fields using a set view, shown in Figure 1. Since the `transient` keyword is meaningful only for serializable classes, only fields from such classes are taken into account here. The set of fields ($\mathbb{F}$) is divided between fields that should be transient ($\mathbb{T}$) and those which should not ($\overline{\mathbb{T}}$). Among $\mathbb{T}$, some of the fields should be transient because they store references ($\mathbb{T}_R$, $\mathbb{T}_R \subset \mathbb{T}$). Technically, in the source code such references can be encoded inside object references but also in

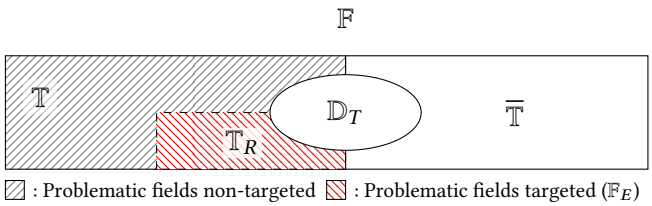

Figure 1: Set view of targeted problem

`long` or `int`. If a field is typed as an `int` or a `long` it is difficult to determine if it stores a reference or simply a value, and thus needs to be declared transient by the developer.

As a consequence, the developer has to declare the fields that are transient ($\mathbb{D}_T$). The fields declared transient should be equal to the fields that should be transient ($\mathbb{D}_T = \mathbb{T}$). However, the developer sometimes forgets to declare transient fields: $\mathbb{T} \setminus \mathbb{D}_T \neq \emptyset$. More dangerously, if the forgotten field should be transient because it stores a reference ($\mathbb{T}_R \setminus \mathbb{D}_T \neq \emptyset$), this omission might make the entire application vulnerable to serialization attacks [8]. This set of fields is named exploitable fields ($\mathbb{F}_E = \mathbb{T}_R \setminus \mathbb{D}_T$). We present a solution to automatically detect exploitable fields ($\mathbb{F}_E$) *i.e.* non-declared-transient reference fields, providing therefore the developers with a watchdog able to warn them before releasing their application.

Generic approach. Practically, to obtain $\mathbb{F}_E$, the analysis has to compute $\mathbb{D}_T$ and $\mathbb{T}_R$. The set of declared fields ($\mathbb{D}_T$) is easily obtained by looking at the Java definitions of fields. On the other hand, computing the set of reference fields ($\mathbb{T}_R$) requires for each field to determine whether it encodes a memory address or not. As the type is not enough to decide, the usage of this field inside the code needs to be tracked. If the field stores a reference then it should be used at some point as a pointer. We propose, here, a static method to track the references employed in C code inside the Java code using taint analysis. The difficulty of our approach lies in the duality of Android applications: while the Java code declares object-level fields, the C/C++ part can deal with them as low-level memory locations.

The overall architecture of our solution is given in Figure 2. First, the C/C++ code is analyzed to list the fields that are used as pointers. This constitutes the first part of the reference fields ($\mathbb{T}_{R_1}$). In addition, the C/C++ analyzer lists all the pointers that interface with the Java code. Using this list, the Java analyzer conducts its own taint analysis to track all the fields that interact with these pointers. This allows to retrieve the second part of the reference fields ($\mathbb{T}_{R_2}$). Finally, the declared fields ($\mathbb{D}_T$) are extracted and subtracted from the reference fields ($\mathbb{T}_{R_1} \cup \mathbb{T}_{R_2}) \setminus \mathbb{D}_T$, in order to compute the set of exploitable fields ($\mathbb{F}_E$).

```
public transient long referenceField;
JNIEXPORT void JNICALL native_method(JNIEnv* env, jobject thisObj) {
   jclass cid = env->FindClass("ThisClass");
   jfieldID fid = env->GetFieldID(cid, "referenceField", "L");
   unsigned long ptr;
   ptr = (unsigned long) malloc(sizeof(char));
   env->SetLongField(thisObj, fid, ptr);
}
```

Listing 2: Reference fields in source code: Native only assignment

```
public transient long referenceField;
JNIEXPORT void JNICALL native_method(JNIEnv* env, jobject thisObj) {
   jclass cid = env->GetObjectClass(thisObj);
   jfieldID fid = env->GetFieldID(cid, "referenceField", "L");
   unsigned long ptr = env->GetLongField(thisObj, fid);
   free((void*)ptr);
}
```

Listing 3: Reference fields in source code: Native only usage

```
public transient long referenceField;
JNIEXPORT jlong JNICALL native_method(JNIEnv* env, jobject thisObj) {
   return (unsigned long) malloc(sizeof(char));
}
public void java_method() {
   this.referenceField = this.native_method();
}
```

Listing 4: Reference fields in source code: Native to Java assignment

```
public transient long referenceField;
public void java_method() {
   this.native_method(this.referenceField);
}
JNIEXPORT void JNICALL native_method(JNIEnv* env, jobject thisObj, jlong arg) {
   free((void*)arg);
}
```

Listing 5: Reference fields in source code: Java to native usage

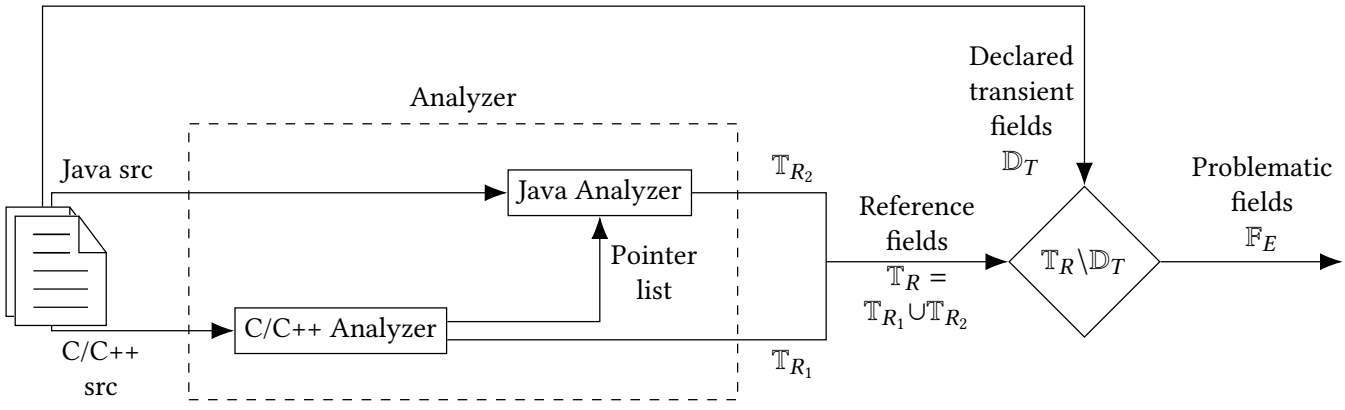

Figure 2: Architecture overview

4 REFERENCE FIELDS DETECTION THROUGH TAINT ANALYSIS

4.1 Reference field patterns in source code

In order to determine precisely which taint analysis to conduct, we have listed all possible implementations of reference fields ($\mathbb{T}_R$). Like every variable or field, a reference field can be either set or used. Because only native code is able to handle raw memory addresses, assignment or usage can happen either directly in native code or within a mix between native and Java code. This results in four distinct code patterns. These patterns are shown in Listings 2, 3, 4, 5 and are the following:

(1) **Native only assignment**: an address, or pointer in C/C++, is used to set a field through JNI. For example in Listing 2, the address retrieved from `malloc` at Line 6, is set to the field `referenceField` of ID `fid` at Line 7 using the JNI `SetLongField` method.
(2) **Native only usage**: a value obtained from a field through JNI is casted to a pointer. For example in Listing 3, the field value retrieved Line 5 using JNI `GetLongField` method, is casted to a pointer at Line 6.
(3) **Native to Java assignment**: an address that is returned by a native method is assigned to a field in a Java method. For example in Listing 4, the address returned by the C++ `native_method` at Line 3 is assigned to the field `referenceField` by Java code at Line 6.
(4) **Java to native usage**: a field is used, in a Java method, to initialize an argument of a native method that casts this argument to a pointer. For example in Listing 5, the field `referenceField`, is used as an argument when calling the C++ `native_method` at Line 3. During this call, the field is casted to a pointer at Line 6.

Focusing on the aforementioned patterns, we have the guarantee that our approach will extensively retrieve all the reference fields ($\mathbb{T}_R$) and therefore successfully build the list of fields whose `transient` keyword is missing ($\mathbb{F}_E$).

4.2 Pattern detection using taint analysis

In order to detect the patterns presented in the previous section, we compute information flows where sources and sinks are associated to fields or memory pointers. All possible sources and sinks, i.e. the elements of the native or Java source code that may produce a flow, are summarized in Table 1. For example, a "Native only assignment" flow exists if the native code manipulates a pointer (source) and calls a JNI Set field method using this pointer in its argument (sink). For patterns that are composed of Java and C/C++ code, the information flow is more complex as two parts (or more) of the code have to be analyzed together. For example, a "Native to Java assignment" flow starts in the native code when a pointer is used and returned by a method called in the Java code, and ends in the Java code when the returned value is set to a field. This is symbolized by a star in Table 1 and we call this a "tunneled source/sink" because the sink of the native code becomes a source for the Java code. When a taint reaches a tunneled sink, in a given programming language, it does not report a problematic flow but instead creates a new tunneled source, in the other language.

All information flows associated to assignment/usage patterns are illustrated in Figure 3. When analyzing the code, the sources and sinks are created using the following rules:

(1) **Native only assignment**: all pointers, *i.e.* all values whose type contains "`*`" or any cast to such a type, are considered as sources. All JNI `Set*Field` methods are considered as sinks. Thus, in Listing 2, Line 6 generates a taint that is propagated until the sink Line 7.
(2) **Native only usage**: all field values retrieved using a JNI `Get*Field` method are sources and all casts to the pointer type are considered as sinks. Thus, in Listing 3, Line 5 generates a taint that is propagated until the sink Line 6.
(3) **Native to Java assignment**: in native source code, every pointer is considered as a source and return operations are tunneled sinks. Then, in Java source code, every return of a method corresponding to tunneled sinks is a tunneled source. The sinks are all the field assignments that happened in Java. Thus, in Listing 4, Line 3 generates a taint (`malloc` returns a pointer) and sinks this taint since it is a return operation. Then, Line 6 generates another taint, because `native_method` is a tunneled source. Finally, the same Line 6 sinks the taint during the assignment.
(4) **Java to native usage**: in Java source code, all field values are sources generating taints that may be sunk passing through native method arguments. Then, in the native source code of these specific methods, the argument that has sunk a field value is a tunneled source. The taint finally sinks when reaching a cast to the pointer type. Thus, in Listing 5, Line 3 propagates the taint of the field `referenceField` to the first argument of the `native_method`. Thus, the third argument Line 5 generates a taint that is propagated until the cast Line 6.

Thus, a tool which correctly conducts all these taint analyses is able to detect any pattern of usage, in other words any reference field ($\mathbb{T}_R$).

4.3 Analysis architecture

Practically, our taint analysis has been implemented following the architecture presented in Figure 2. The source code is first split between Java and C/C++ and each language is handled by its own analyzer which are described in this section.

C/C++ analyzer. For the C/C++ part, the taint analysis is built over clang [5]. Clang provides an event-driven API for developing static analyzers. After converting the C/C++ source code into an Abstract Syntax Tree (AST), clang offers the possibility to call hooks before or after specific expression evaluation. Our hooks are reported in Algorithm 1.

For optimization purposes, all four taint analyses are conducted at the same time with three different types of taint: **A** for the method arguments; **F** for the fields got using JNI; **P** for the pointers. Taints are applied to C/C++ expressions. If no taint has been applied to an expression, then this expression is considered to carry taints of its sub-expression (*e.g.* `a+b` carries the taints of `a` and the ones of `b`).

Finally, the C/C++ analyzer outputs a JSON file that contains:

(1) a list of fields that have been detected through "native only assignment" and "native only usage" patterns;

Table 1: Sources and sinks for detecting reference field patterns

Patterns		Native source code		Java source code	
		Sources	Sinks	Sources	Sinks
Native only	assignment	Pointers	JNI Set fields	∅	∅
	usage	JNI Get fields	Casts to pointer	∅	∅
Native to Java	assignment	Pointers	Return operations*	Native method returns*	Field assignments
Java to native	usage	Method arguments*	Casts to pointer	Fields value	Native method arguments*

*: tunneled source/sink.

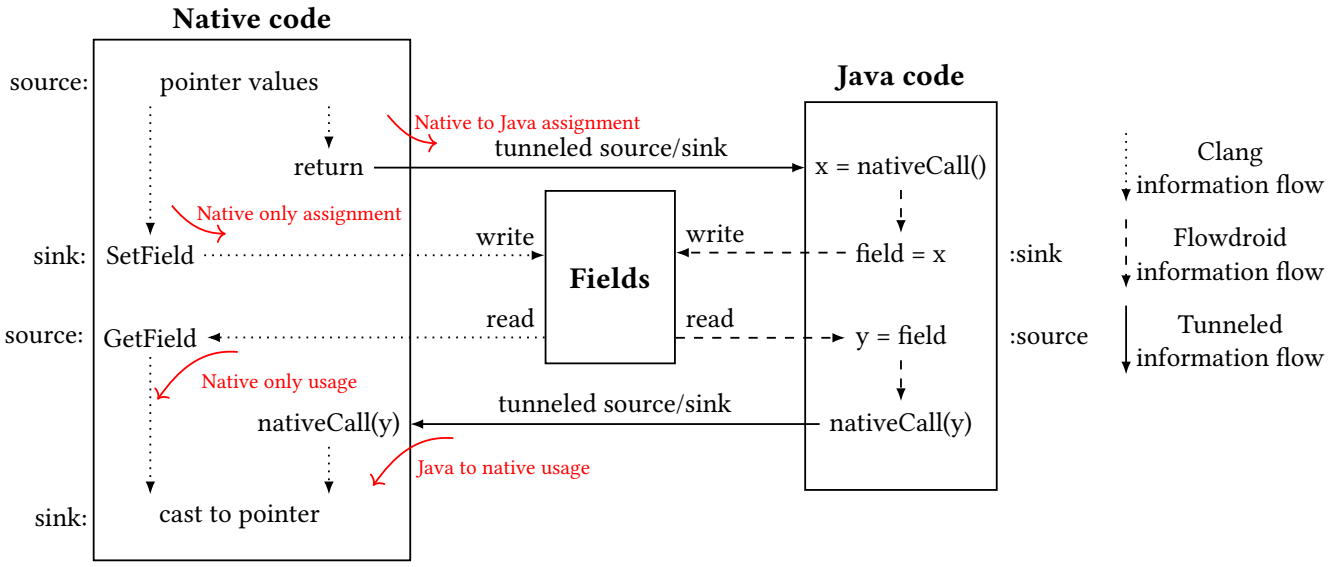

Figure 3: Representation of reference flow tracking

(2) a list of methods whose return values are pointers. This corresponds to the native part of the "native to Java assignment" pattern;
(3) a list of method arguments that are used as pointers that coincide with the "Java to native usage" pattern.

Java analyzer. For the Java part, the taint analysis is built over Flowdroid (version 2.7.1) [1]. Flowdroid provides the same event-driven programming interface as clang. It also gives an additional class hierarchy lookup mechanism that is used to filter and treat only fields from serializable classes. The Java analyzer takes as input the Java code, the list of methods and the list of method arguments generated by the C/C++ analyzer. Using these lists to set up its taints and sinks, Flowdroid generates a list of fields detected through "native to Java assignment" and "Java to native usage" patterns. Here no special taint management is performed since the two taint analyses are not mixed.

Final reporting. Finally, a union is made between the two field lists generated by respectively the C/C++ and Java analyses. This set is the set of the reference fields ($\mathbb{T}_R$). By parsing the code, the fields declared as transient ($\mathbb{D}_T$) are retrieved and the potentially exploitable fields set $\mathbb{F}_E$ is computed by subtracting the set of declared transient fields from the set of reference fields ($\mathbb{T}_R \backslash \mathbb{D}_T$).

4.4 Static analysis limitations

Due to its static nature, the analysis suffers from common static Android taint analysis drawbacks. In particular, when a field is manipulated through reflection, the analysis cannot determine which field is used and so cannot report a potential missing `transient` keyword, leading to false-negative generation.

Moreover, the Java analysis has to match the name of the native method in the C/C++ source code to the one in the Java source code. Even if the Android native method loader uses a convention for the native method naming, developers can register their own names by using the JNI method `RegisterNatives`. As it is done at execution time, this behavior cannot be retrieved by the static analysis and may generate false positives and negatives. However, the developer could inform about his specific mappings.

5 EXPERIMENTAL VALIDATION

In order to evaluate and validate our novel approach, we need to analyze the full source code of applications. To get a chance to find vulnerabilities related to serialization, these applications should have native code and additionally, should manipulate objects from both sides. Finding such open source applications is very difficult,

```
1 Source: <OpenSSLX509Certificate: long mContext>
2 Sink: $l4 = staticinvoke <NativeCrypto: long X509_get_notAfter(long,OpenSSLX509Certificate)>($l3, r0)
3 Source: <OpenSSLX509Certificate: long mContext>
4 Sink: $l2 = staticinvoke <NativeCrypto: long X509_get_notBefore(long,OpenSSLX509Certificate)>($l1, r0)
5 Source: <OpenSSLX509Certificate: long mContext>
6 Sink: staticinvoke <NativeCrypto: void X509_free(long,OpenSSLX509Certificate)>($l2, r0)
```

Figure 4: `OpenSSLX509Certificate` Java analysis output

Algorithm 1 C/C++ analyzer taint management

```
on event after function call do
  if called function is JNI Get Field then
    Result expression is tainted with taint F
  end if
end event

on event before value declaration do
  if value is an argument of the analyzed function then
    Value expression is tainted with taint A
  else if value is a variable then
    Value expression is tainted with taint P
  end if
end event

on event before function call do
  if called function is JNI Set Field then
    if second argument of the called function is tainted with P
    then
      log native only assignment
    end if
  end if
end event

on event before function return do
  if return value expression is tainted with P then
    log native to java assignment
  end if
end event

on event before casting expression do
  if expression is casted to a pointer then
    if casted expression is tainted with F then
      log native only usage
    end if
    if casted expression is tainted with A then
      log java to native usage
    end if
  end if
end event
```

as most candidate applications from the Google Play store do not release their source code. Moreover, by construction of our approach, systematic testing is not easily automatable as the recompilation process needs fine-tuning. Thus, large-scale benchmarking of our approach becomes unrealistic.

Nevertheless, we performed the following tests that clearly show the benefit of the proposed methodology. We analyzed three cases:

(1) Constructive validation: we construct an application that regroups the four patterns described in Listings 2, 3, 4, 5. These cases can be seen as unit tests that confirm that the tool works as expected.
(2) Literature confirmation: we review the `OpenSSLX509Certificate` class from the conscrypt[3] library, as according to Peles & Hay [8], this class contains a field named `mContext` which should be vulnerable for not being transient. This test shows that our tool can reproduce previous results automatically, enhancing the results of Peles *et al.* who discovered the vulnerabilities manually.
(3) At-scale verification: we select, based on its popularity and its robustness, the Telegram application[4] as it constitutes a large open-source Android application with more than 1 million lines of code spread across Java and C++. This test shows the benefit of our tool when analyzing the full source code of an application.

In a nutshell, instead of seeking exhaustivity, our experimental protocol focuses on validating our approach and aims to show that: *a*) our patterns are catchable; *b*) the previously known vulnerabilities are retrieved automatically; and *c*) our solution can be used to check large open-source applications. Finally, on a different dimension, we also discuss the performance of our approach in terms of computational time and resource consumption.

5.1 Constructive validation of the patterns

As intended, all four transient fields have been detected by their respective pattern. For patterns "native to Java assignment" and "Java to native usage", the analysis has logged the transient field names and their class names. It also logged the name of the native method responsible for setting or using the transient field. For "native only" patterns, the analysis only logged the name of the transient fields. It also recovered the class name for native only assignment but did not manage for the native only usage pattern. This pattern, see Listing 3, uses the JNI method `GetObjectClass` (line 3) instead of `FindClass` whose argument is the class name. As mentioned in Section 4.4, this method is not handled yet.

5.2 Catching known errors

When running the C/C++ analyzer over the `OpenSSLX509Certificate` class source code, no fields were reported using the native only patterns. For the "native to Java assignment" pattern, the C/C++ analyzer has reported 127 native methods whose return value is a pointer, that is 127 tunneled sinks. On the Java side, the analyzer has not reported any corresponding information flow

[3] https://github.com/google/conscrypt
[4] https://github.com/DrKLO/Telegram, tag: release-5.15.0_1869

Table 2: Non-vulnerable flows reported for Telegram

Reported fields $\mathbb{F}_E$		Native to Java assignment	Java to native usage	Native only patterns
Class name	Field name			
`tgnet.TLObject`	`ip ipv6 peer_tag`			✓
`messenger.BaseController`	`currentAccount`	✓		
`messenger. NotificationCenter`	`currentAccount`	✓		
`SQLite.SQLiteDatabase`	`sqliteHandle`	✓	✓	
`SQLite. SQLitePreparedStatement`	`sqlite StatementHandle`		✓	
`tgnet.NativeByteBuffer`	`address`	✓		
`ui.Components. RLottieDrawable`	`nativePtr`	✓	✓	

Table 3: Analysis time

Name	Java		C++	
	SLOC	duration	SLOC	duration
Patterns example	26	1.508s	42	0.34s
`OpenSSLX509`	663	5.499s	8,269	356.93s
Telegram	511,519	6min 6s	628,864	5h 02min

because the returned values are used for initializing fields that are object references. Object references do not need to be declared transient, *cf.* Section 2. For the "Java to native usage" pattern, the C/C++ analyzer has reported 111 native method arguments treated as pointers inside the native code, *i.e.* 111 tunneled sources. The Java analyzer has reported 3 corresponding information flows, all related to the field `mContext`, which is the one that was previously reported vulnerable [8]. Thus, the analysis has reported no false positives and has detected the three vulnerabilities.

The source sink couples are reported in Figure 4. As shown in this figure, the fields are clearly identifiable and the output could be read by any developer even without specific security-oriented knowledge.

5.3 Checking a large application

The flows reported during the analysis of Telegram are reported in Table 2. In order to assess that the analysis scales with huge codebases, we have analyzed all classes including non-serializable ones (even if `transient` keyword is only meaningful in serializable classes). The C/C++ analysis of Telegram has reported three fields using the "native only" pattern: `ip`, `ipv6`, `peer_tag`. For all these fields the class name was not recovered since the C/C++ code uses `GetObjectClass` to retrieve the class of the accessed fields. By manually looking at the source code, we found that the three fields are declared several times in subclasses of the `tgnet.TLObject` class. The three fields are `String` fields: they are references to objects, which are handled by the serialization process. Thus, they do not need to be declared transient and are *false positives*. For the "native to Java assignment" pattern, the C/C++ analyzer has reported 26 native methods whose return value is a pointer (tunneled sinks). The Java analyzer then reported 5 fields that should be transient. On the other hand, for the "Java to native usage" pattern, the C/C++ analyzer has reported 62 method arguments that are treated as pointers (tunneled sources) which has led the Java analyzer to report 3 fields. Since 2 fields are reported by both patterns, we finally obtained 6 programming flows of `int` or `long` fields that should be transient. Nevertheless, the 6 corresponding declaring classes are not serializable. As a consequence, the forgotten `transient` keywords do not lead to vulnerabilities, in the current state of Telegram. The results reported are programming errors that could bring vulnerabilities if a developer updates these classes into making them serializable.

5.4 Analysis time

We have recorded the time elapsed during the analysis of the three cases (non-serializable classes omitted for Telegram analysis since `transient` keyword is only meaningful in this context). The times and the number of Source Line Of Code (SLOC) analyzed are reported in Table 3. Analyses have been run using 26G of DDR4 RAM and an Intel Core i7-8850H (12 threads, 2.60GHz) processor. Even for the huge Telegram application (encompassing more than 1 million lines of code), the analysis terminates but takes five hours. The proposed method is not intended to be used frequently during the development cycle but rather only once before the application release, as part of the continuous integration tests.

6 CONCLUSION

This article presents a novel method to detect forgotten `transient` keywords in Android applications. Our method is fully automated, analyzing both native code and Java bytecode. It relies on computing flows between sources and sinks that may spread data that should have been transient. Despite the complexity of the code that switches between native and Java, we are able to reliably detect flows, considerably enhancing the previous results of Peles *et al.* [8] that pointed out manually this kind of vulnerability.

The experimental evaluation of our tool faces the problem of the availability of the full source code of Android applications that may suffer from such vulnerabilities. Only very few applications that would be candidates can be found on open source repositories. Nevertheless, we studied two cases. First, we confirmed the results of Peles *et al.* on the OpenSSL X509 certificate library. Second, we studied the Telegram application, as its source code is available on Github. We discovered flows in the source code. These flows are transient fields that have been forgotten but that cannot be exploited to leak sensitive data. We also discussed the practical performance of our tool and we show that it could be used in any Android application releasing process.

In future work, we plan to apply our method to review other native entities, such as sockets or file descriptors, that may also create security holes if leaked into Java code. The opposite phenomenon, *i.e.* Java values leaking into native code, could be sources of vulnerabilities, such as usage of unsanitized user input. Such attacks are complex to detect statically and may be of interest during an application's source code review. Thus, both exploitability and detection possibilities of such entities should be explored.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[2] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *International conference on Mobile Systems, Applications, and Services*. ACM, Bethesda, Maryland, USA, 239–252.

[3] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *11th USENIX conference on Operating systems design and implementation*. USENIX, Vancouver, Canada, 393–407.

[4] Jann Horn. 2014. CVE-2014-7911: Android <5.0 Privilege Escalation using ObjectInputStream. seclists.org/fulldisclosure/2014/Nov/51

[5] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5.

[6] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *ACM conference on Computer and Communications Security*. ACM, Raleigh, North Carolina, USA, 229–240.

[7] Ibtisam Mohamed and Dhiren Patel. 2015. Android vs iOS security: A comparative study. In *International Conference on Information Technology-New Generations*. IEEE, Las Vegas, NV, USA, 725–730.

[8] Or Peles and Roee Hay. 2015. One Class to Rule Them All: 0-day Deserialization Vulnerabilities in Android. In *USENIX Workshop on Offensive Technologies (WOOT 15)*. USENIX, Washington, D.C., USA.

[9] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin TS Chan. 2014. On tracking information flows through JNI in Android applications. In *IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, Atlanta, USA, 180–191.

[10] Mingshen Sun, Tao Wei, and John Lui. 2016. TaintART: A practical multi-level information-flow tracking system for Android RunTime. In *23rd ACM SIGSAC Conference on Computer and Communications Security*. ACM, Vienna, Austria, 331–342.

[11] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM, Toronto, Canada, 1137–1150.

[12] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. 2018. NDroid: Toward tracking information flows across multiple Android contexts. *IEEE Transactions on Information Forensics and Security* 14, 3 (2018).

[13] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *USENIX Security Symposium*. USENIX, Vancouver, Canada, 289–306.

[14] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. 2014. IntentFuzzer: detecting capability leaks of android applications. In *ACM symposium on Information, Computer and Communications Security*. ACM, Kyoto, Japan, 531–536.

[15] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. 2017. Taintman: An art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices. *IEEE Transactions on Dependable and Secure Computing* 17, 1 (2017).