OATs'inside: Retrieving Object Behaviors From Native-based Obfuscated Android Applications

PIERRE GRAUX, Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, France
JEAN-FRANÇOIS LALANDE, VALÉRIE VIET TRIEM TONG, and PIERRE WILKE,
CentraleSupélec, Inria, CNRS, University of Rennes, IRISA, France

Analyzing Android applications is essential to review proprietary code and to understand malware behaviors. However, Android applications use obfuscation techniques to slow down this process. These obfuscation techniques are increasingly based on native code. In this article, we propose *OATs'inside*, a new analysis tool that focuses on high-level behaviors to circumvent native obfuscation techniques transparently. The targeted high-level behaviors are object-level behaviors, i.e., actions performed on Java objects (e.g., field accesses, method calls), regardless of whether these actions are performed using Java or native code. Our system uses a hybrid approach based on dynamic monitoring and trace-based symbolic execution to output control flow graphs (CFGs) for each method of the analyzed application. CFGs are composed of Java-like actions enriched with condition expressions and dataflows between actions, giving an understandable representation of any code, even those fully native. *OATs'inside* spares users the need to dive into low-level instructions, which are difficult to reverse engineer. We extensively compare *OATs'inside* functionalities against state-of-the-art tools to highlight the benefit when observing native operations. Our experiments are conducted on a real smartphone: We discuss the performance impact of *OATs'inside*, and we demonstrate its practical use on applications containing anti-debugging techniques provided by the OWASP foundation. We also evaluate the robustness of *OATs'inside* using obfuscated unit tests using the Tigress obfuscator.

CCS Concepts: • **Security and privacy** → **Malware and its mitigation**; **Mobile platform security**; **Software reverse engineering;**

Additional Key Words and Phrases: Android native application, trace based symbolic analysis, obfuscation

ACM Reference format:
Pierre Graux, Jean-François Lalande, Valérie Viet Triem Tong, and Pierre Wilke. 2023. *OATs'inside*: Retrieving Object Behaviors From Native-based Obfuscated Android Applications. *Digit. Threat. Res. Pract.* 4, 2, Article 30 (August 2023), 27 pages.
https://doi.org/10.1145/3584975

1 INTRODUCTION

Analyzing Android applications is a crucial task for security experts to ensure security for mobile phone users. These experts must quickly understand the main purpose of the applications and even their hidden features. They have to assess whether the applications violate a given security policy or whether they are indeed malware. Meanwhile, both legitimate developers of proprietary code and malware authors try their best to avoid such

Authors' addresses: P. Graux, Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France; email: pierre.graux@univ-lille.fr; J.-F. Lalande, V. Viet Triem Tong, and P. Wilke, CentraleSupélec, Inria, CNRS, University of Rennes, IRISA, 35000 Rennes, France; emails: {jean-francois.lalande, valerie.viettriemtong, pierre.wilke}@centralesupelec.fr.

2576-5337/2023/08-ART30
https://doi.org/10.1145/3584975

analyses. Developers use various obfuscation techniques whose sole purpose is to prevent, limit, and slow down reverse analyses.

These obfuscation techniques usually target the code, which is the primary source of information about what a software can perform. In Android applications, the code can be composed of both Dalvik bytecode [3], potentially compiled into an OAT file, and native code. Wong and Lie [34] identified three main classes of obfuscation techniques. First, language-based obfuscation techniques take advantage of features provided by the bytecode. This includes, for example, Java reflection, value encryption, and dynamic code loading. Second, a developer may also eschew Java by calling native code and relying entirely on classical techniques used by desktop software such as code packing, virtualization, anti-debug and emulation operations, or self-modifying code [11]. These obfuscation techniques are full-native code techniques. Last, runtime-based obfuscation techniques go further and allow the obfuscated code to alter the conventional Android system behavior. Thus, analysis tools can no longer rely on assumptions about the **Android Runtime (ART)** functioning. For example, a program can modify the target of a call or modify the executed bytecode on the fly.

One of the most used and developed runtime-based technique is application packing [13], which consists in storing the bytecode encrypted in the application and only decrypting and loading it at execution time by modifying the runtime internal structures instead of simply using the class loading facilities provided by the Java language. This prevents static analysis tools from accessing the bytecode. Recently, new techniques that consist in compiling sensitive bytecode methods to assembly code have been developed [8, 15]. When such advanced obfuscation techniques are involved, the bytecode is never available for the analyst. For these reasons, an analyst cannot rely only on static analysers outputs that would miss the sensitive classes obfuscated by the developer.

Dynamically reversing an application obfuscated using runtime-based techniques gives other challenges. First, examining the native code is a more complex task than reversing the bytecode: Abstractions such as types, object hierarchy, and function boundaries are unavailable in native code. Second, the native code can interact with Java objects through a dedicated interface called **Java Native Interface (JNI)**. These interactions can be easily observed [14, 19, 24], but an obfuscated code may bypass this interface, because the native code has full rights to modify the process' memory [15]. For this reason, most state-of-the-art tools, which almost all rely on a clearly defined interface between bytecode and native code, may miss some operations performed by the native code. A general method for observing the actions performed by the native code, independently from the JNI interface, would help the analyst to observe the natively obfuscated operations done at runtime that would have an impact on Java objects.

All the aforementioned obfuscation techniques take advantage of (1) the possible interleaving between bytecode and native code during the execution of an application and (2) the fact that most static analysis or deobfuscation tools are currently only able to handle one language at a time. As creating a new deobfuscator handling two languages would be intractable, we propose to focus on the observation of the Java environments while letting the native code, even obfuscated, be executed.

This article presents *OATs'inside*, an open source[1] stealth analyzer recovering **object-level CFGs (olCFGs)** of Android applications. An olCFG is a graph composed of actions performed on Java objects (e.g., field accesses and method calls), regardless of whether these actions are performed using Java or native code. These olCFGs are retrieved even in the presence of obfuscation techniques targeting native code. Note that *OATs'inside* does not target the recovering of all native operations occurring in an obfuscated native code but only focuses on operations that affect the Java environment. *OATs'inside* relies on a combination of static analysis, instruction hooking, and multiple executions to build iteratively the action performed by the native code on the Java objects. *OATs'inside* builds a olCFG operating on Java objects even if these objects have been altered by native code. Each node represents one of the following actions: a field access, a method call, an exception, or a monitor session. To obtain these olCFGs, *OATs'inside* combines two analyses: a modified ART that logs every object-level event

[1]Release url: https://gitlab.inria.fr/cidre-public/oatinside.

made by the application for generating a olCFG and a symbolic execution analysis that enriches this olCFG with expressions and conditions related to the variables manipulated by actions. With the results of the outputted olCFG, *OATs'inside* can be launched again with new dynamic inputs. This allows us to extend the olCFG by iteratively discovering the full behavior of an application.

We extensively compare the functionalities of *OATs'inside* with state-of-the-art Android analysis tools to show the better coverage of *OATs'inside* with native operations impacting Java objects. We showcase a typical usage of *OATs'inside* on an application from a collection of mobile reversing challenges provided by the OWASP foundation. We successfully solve the challenge, whereas the application contain anti-debugging techniques. We evaluate *OATs'inside* functionalities using unit tests. We also obfuscated these unit tests using the Tigress obfuscator [10], and we discuss the limits of *OATs'inside* on these obfuscated unit tests. We evaluate the performance of *OATs'inside* on a worst-case scenario where an AES-128 algorithm is executed natively but interacts with Java objects to store intermediate results. We also show that the overhead can be reasonable using the OWASP challenge compared to the worst-case scenario.

The contributions of this article are the following:

- *OATs'inside*, a tool for computing an olCFG representing operations on Java objects even if these objects have been altered by native code;
- a new method for mixing traces and symbolic executions in an unusual way: The values obtained during the executions are always used to drive the symbolic execution instead of classically using an SMT solver. Additionally, this allows us to annotate the traces using symbols for further understanding;
- unit tests and their obfuscated versions that exercise all Java operations combining primitive types and object fields for evaluating the completeness of our approach.

The rest of the article is organized as follows. Section 2 gives the necessary background to understand the challenges involved when analyzing Android applications, in particular the general methods for obfuscating the code with native-based methods. Then, the related work is presented in Section 3. Section 4 explains how *OATs'inside* retrieves object-level behaviors of obfuscated applications. Section 5 provides the readers with the implementation details. Section 6 reviews and discusses the results and findings of the conducted experiments. Finally, we draw our conclusions in Section 7.

2 BACKGROUND

This section depicts the necessary background on what composes the code of an Android application and how runtime-based obfuscation operates to hide their codes. Even if we present some historical elements on the evolution of Android internals, all the technical details are strictly relative to version 7.0 (Nougat, 2016) of Android and we focus on the ARMv8 [4] architecture as it is the most recent version of the most used architecture [29]. This version has been chosen because we run our experiments on a real smartphone: the Sony Xperia X. This smartphone is one of the first devices to be part of the Open Devices program, which simplifies the recompilation of the AOSP tree.[2] The version 7.0 uses Kernel 3.10 but can be easily adapted to version 7.1 that runs kernel 4.4. We also believe that most of the discussed internals also apply to more recent versions of Android. Nevertheless, we did not port *OATs'inside* to these recent versions yet, because this would require development costs and would not bring new scientific results to our experiments.

2.1 Android Application Execution

Application and DEX format. An Android application is an archive composed of a *Manifest* file (metadata), resource files (texts, pictures, and layouts), and a file that contains the application bytecode in the DEX format [2]. Dalvik is the virtual machine that interprets and performs **just-in-time (JIT)** compilation of the application

[2]https://developer.sony.com/develop/open-devices/get-started/supported-devices-and-functionality/.

bytecode written for Android. The DEX bytecode is independent of the device architecture. Since Android 4.4 (KitKat), Dalvik bytecode can be compiled into assembly code but can still be interpreted by the ART. Hence, analysis tools cannot avoid handling Dalvik bytecode.

Ahead-of-Time Compilation. Since Android 5.0 (Lollipop, 2014), the bytecode of an application can be compiled into native assembly code. Forcing the compilation of the bytecode of every application may cause a huge overhead, especially when the operating system is updated. Thus, two major releases later, in Android 7.0, ART was able to separately compile the methods: When the method execution frequency reaches a threshold, the system compiles it. The compilation output is an OAT file,[3] which contains the original DEX file and the assembly code of compiled methods [27]. Compiling the methods independently solves the overhead problem during the first launch of an application; however, it complexifies the execution at runtime. Indeed, the executed code often switches between compiled and bytecode methods. Consequently, this modification increases the work of analysis tools and makes obsolete the previous approaches that rely on instrumenting the DEX interpreter [14, 17, 38].

Native Code. For performance purposes, Android allows developers to create parts of applications in C or C++. When compiled, the resulting code is assembly code, saved in an independent native library file. Hereinafter, we will call *compiled bytecode* and *compiled C/C++ code* the AOT code produced by the compilation of the bytecode and the native code produced by C/C++ code, respectively. The introduction of native code can be used by obfuscation techniques and thus complicates analyses.

Runtime and Application Execution. The switches among Dalvik bytecode, compiled bytecode, and native code are managed by the runtime library. Additionally, this library is in charge of virtual machine aspects such as **garbage collector (GC)**, JIT compiler, and interpreter. When an application is executed, the runtime library maps the different sections of the OAT file and its associated native libraries into the memory. The DEX file and the assembly code are copied in the data, respectively executable, section of the memory.

When executing a method, the runtime library preferentially executes assembly code, if available, rather than bytecode. Because the bytecode is not recognized by the underlying architecture, it cannot be directly executed on the phone but is rather JITted or interpreted, depending on the execution statistics: A frequently executed method will be JITted rather than be interpreted. Consequently, advanced obfuscation techniques can modify the bytecode or the assembly code. Observing the effects on the manipulated objects of the heap becomes a challenging task because of the heterogeneous software components of the application.

Heap manipulation. Each type of code (bytecode, AOT, or native) acts on the heap differently. First, heap accesses originating from the bytecode are performed by the virtual machine. Second, heap accesses originating from the assembly code in the OAT file are directly performed with load and store assembly instructions. As a result, the typing information is lost. Third, heap accesses originating from native code are managed through the JNI. Going through the JNI is necessary, because the native code is not compiled for a specific Android version, and, therefore, field offsets are not known at compile time.

One of our objectives is to monitor the object accesses performed by the application. Thus, we will need to deal with these three types of heap access. As detailed later in Section 3, the heap accesses performed by the virtual machine or the JNI are straightforward to manage, because the virtual machine or the JNI can log these accesses. Direct heap accesses are more difficult to monitor because of the loss of typing information. This problem was partially addressed by Xue et al. [37]; we provide a complete solution to this challenge in Section 4.

2.2 Runtime-based Obfuscation

An obfuscation technique refers to any means of complicating the quick understanding of the application code by a manual or automatic process. Applying an obfuscation technique should produce hard-to-understand

[3]We could not find an official definition for this acronym.

code, which perfoms the same functionalities as the original code. Similarly to Xue et al. [8], we distinguish language-based, full-native, and runtime-based obfuscation techniques. Full-native obfuscation methods have been extensively studied [11, 21, 26, 31]. Runtime-based techniques exploit the possibilities offered by ART and the inter-connection between native and Java worlds to obfuscate code. These obfuscation methods, namely *native DEX packing*, ***ahead-of-time compilation– (AOTC)*** *based bytecode hiding*, and *direct heap access*, are presented below.

Native DEX packing methods. Packing consists in storing the Dalvik bytecode ciphered, which will be dynamically deciphered and loaded at runtime. This relies on native code that directly modifies Android's internal structures for deploying new bytecode. These modifications can occur at any time of the bytecode loading process or before the execution. Packing has been extensively studied [20, 34, 36, 39, 40], and packing DEX bytecode using native code is a popular technique: Duan et al. [13] reported that an average of 13.89% of malware in the wild between 2010 and 2015 used packing techniques to hide malicious behavior.

AOTC-based bytecode hiding. The AOTC-based bytecode hiding scheme is a recently described obfuscation technique [8] that aims at hiding the bytecode of sensitive methods from both static and dynamic analyses. It is composed of three main steps performed before releasing the APK file. First, the bytecode of obfuscated methods is removed from the DEX file. Then, the bytecode of these methods is compiled using a custom compiler, producing native code. Finally, the compiled code is added to the APK as a library and calls to obfuscated methods are transformed into JNI calls.

Additionally, contrary to the packing methods, the bytecode is never deciphered when using AOTC-based obfuscation and, thus, is never directly available to analysis tools. The code is only present in its compiled form.

Direct Heap Access. **Direct Heap Access (DHA)** consists in using native code to access Java object fields without using JNI [15]. The application directly reads or writes the heap where the field is stored. This allows us to break the well-defined interface between the Java and the assembly worlds and, thus, to circumvent hooking of JNI calls, an analysis technique used by state-of-the-art tools [24, 32, 37] that we describe in the next section.

Before presenting how we tackle these obfuscation techniques in Section 4, we present in the next section the previous approaches that try to recover information from applications obfuscated using the aforementioned techniques, and we explain why they cannot collect the level of information that we intend to capture with our contribution.

3 RELATED WORK

In this section, we focus on previous works that handled native code when dealing with obfuscation techniques. We first describe the works related to packer techniques that try to recover the hidden, dynamically loaded, bytecode, and then the works that try to obtain information about the native code. All the presented works are reported in Table 1, which classifies these works according to four criteria:

- **Object level details**: quality of the information retrieved about the actions that operate at the objects level (e.g., method calls, field accessing, exceptions, allocations) and that are performed by native code. Tools unable to recover any actions are rated "low." Tools able to retrieve partial information about the performed actions are rated "medium." "High" is reserved for tools that retrieve all the details about the performed actions.
- **Nature**: we summarize the nature of the output of the tools.
- **JNI interface agnosticism**: whether the tool relies or not on a clearly defined JNI interface. If so, then the tool may be bypassed by runtime accesses from the native code.
- **Obfuscation robustness**: describes their capacity not to be affected by native-based obfuscation techniques that would hide parts of the code.

Table 1. Evaluation of State-of-the-art Native Android Application Analysis Tools with Respect to the Given Criteria

Tool	Object level details	Nature	JNI Interface agnosticism	Obfuscation robustness
Lantz et al. [18]	low	JNI chain	no	no
Afonso et al. [1]	low	statistics	no	yes
NDroid [24]	medium	flow	no	yes
Malton [37]	medium	flow	no	yes
JN-SAF [32]	medium	flow	no	no
AppLance [19]	low	flow	yes	yes
ArtDroid [12]	low	virtual method call	yes	yes
Ronin [30]	low	libcall	yes	yes
DroidScope [38]	low	ASM	no	yes
Unpackers [34, 36, 39, 40]	low	code loading	yes	tuned
OATs'inside	high	CFG object behaviors	yes	yes

We intend that *OATs'inside* meets all criteria: All object-level actions operated by the native code should be collected, and we expect to observe these actions on a CFG for further manual analysis. Additionally, any action that bypasses the JNI interface should be observed and native-based obfuscation techniques should be handled. We compare later in Section 6.2 these tools to *OATs'inside* with more technical details, when the model of actions that we are able to observe has been presented.

3.1 Unpacking Hidden Bytecode

DexHunter [40] and AppSpear [39] are unpackers that try to discover how native code is used to modify the code loading process. They overload the runtime methods that load the bytecode. When the packed application is launched, the unpacker calls these overloaded methods, which log the loaded bytecode. When the execution is over, the unpackers reconstruct the application by merging the different parts of the bytecode that they obtained.

However, packers can avoid using the functions hooked by the unpackers or even change the bytecode after it is loaded. For this reason, PackerGrind [36] and TIRO [34] do not set up any hooks. Instead, they watch the structures managed by the runtime involved in the bytecode loading process. When these structures are modified, they choose, manually for PackerGrind, to retrieve the newly loaded code. Nevertheless, retrieving the bytecode does not the give information about the actions performed by the native code.

3.2 Information Retrieval for Native Code

Several works have presented generic framework solutions [12, 30, 38] where the analyst can insert some hooking codes to audit native code actions. Indeed, these frameworks can be used to observe virtual method calls [12] (vtable hooking) and library calls [30] (PLT hooking). Lantz and Johansson [18] proposed another static analysis to locate JNI call chains. The output of these tools contains few information; hence, we classify their detail level as low. DroidScope [38] is another framework, based on QEMU, that treats native methods like any other assembly code, and, therefore, it fails to report object-level information. It reports more details such as system calls, memory accesses, and instruction executions. These outputs are too close to assembly, as reported in Table 1.

Many works [19, 24, 32, 37] have aimed at tracking the information flow during the execution of an Android application. NDroid [24] propagates TaintDroid [14] taints by hooking the Android framework methods that call native code and all JNI entry points (QEMU hooking and VM introspection). AppLance [19] hooks source methods and changes their return values. With several executions, it can detect if the sink methods' parameters changed owing to a modification of the source methods' return values. JN-SAF [32] uses angr [28] to statically track information flows. The key idea is to initialize the JNI entry point table with symbolic addresses representing the different JNI methods. Then, JN-SAF can represent their effects symbolically. All these approaches provide all the sensitive information flow between methods, but retrieve only the behaviors involved in the information flow. It is reported as medium in Table 1.

Xue et al. provided a comprehensive view of the Android malware behaviors using Malton [37]. Malton is a dynamic analysis platform that aims to compute information flows. It relies on Valgrind [22] to hook method calls, ART, and framework libraries. It stores the address of every Java and native method and then checks, for every jump, if the destination address is the address of a method. Then Malton reconstructs the Java objects corresponding to the arguments by parsing the memory. It also hooks framework methods responsible for loading code and the JNI entry points and intercepts all system calls. Finally, it propagates taints through every assembly instruction. Moreover, Malton leverages concolic execution to trigger or force the execution of specific, manually tagged, code areas. Like in previously described works [19, 24, 32], the output only focuses on information flow. Moreover, Malton cannot hook methods from the analyzed APK but only the runtime and framework ones.

3.3 Summary

Static approaches [18, 32] can be easily fooled by native code obfuscation. However, unpackers [34, 36, 39, 40] focus on specific techniques for dynamic code loading, but they possibly miss some other ones: While they recover the bytecode that is loaded by the native code, they miss all other actions that may performed. As a result, we seek for a more general approach combining static and dynamic approaches for observing the performed action of the native code.

Three major properties should be ensured when designing our native code analyzers: First, it should not be bypassed by direct heap accesses from native code; second, it should not rely on the JNI interface; third, it should be robust against native-based obfuscations.

4 *OATS'INSIDE*

To address native-based obfuscated Android applications, we describe *OATs'inside*, an Android application analyzer that recovers Java-level operations, even if the application is protected by full-native-based and runtime-based obfuscation techniques. Note that *OATs'inside* does not intend to deobfuscate the native code but aims at observing all possible actions on Java objects even if complex obfuscation techniques have been applied on the native code. *OATs'inside* combines dynamic analysis with symbolic execution. The dynamic analysis gathers sequences of low-level events, and the symbolic execution is driven by these events. *OATs'inside* outputs a CFG that can be passed to existing security analysis tools such as IntelliDroid [33] or directly to a human analyst. The CFG is said to be *at the object level*, because it contains instructions acting on objects such as calling methods or setting object fields. It describes the contents of each method, the conditional expressions involved in the control flow instructions, the dataflow between actions, and the interprocedural calls. The human intervention of the analyst is needed to identify the methods that should be studied. Indeed, depending on the analysis goal and the type of application, the choice of method to study varies. This part cannot be automated and is thus left to the analyst.

OATs'inside adopts a two-step analysis: First, a dynamic analysis, followed by a concolic analysis. Note that this concolic analysis consists in annotating the traces obtained during the dynamic analysis using symbolic names that represent Java objects. These steps are based on four main modules as described in Figure 1. During the dynamic step, the Runner module executes the application and logs every action dealing with objects. As an application requires external inputs, the execution is either driven manually or via a dedicated exploration tool such as Intellidroid [33]. The CFG creator module initializes a first version of the CFG from the actions obtained from the Runner module. During the concolic step, the Concolic analyzer module performs a symbolic execution based on the actions logged by the Runner module and memory snapshots issued by the Memory dumper. It enriches the CFG by recovering conditional expressions at branching nodes and data dependencies between actions.

To illustrate *OATs'inside*'s methodology, we developed `PINtest`, a PIN verification application written in Java. It runs transparently on an Android 7.0 smartphone. We will use this application as a running example throughout the rest of this article. For the sake of readability, we give a simplified version of its source code in Figure 1. If the `pin` field of the calling object (`this`) is negative, then an exception is thrown. `SimpleTestPIN.test`

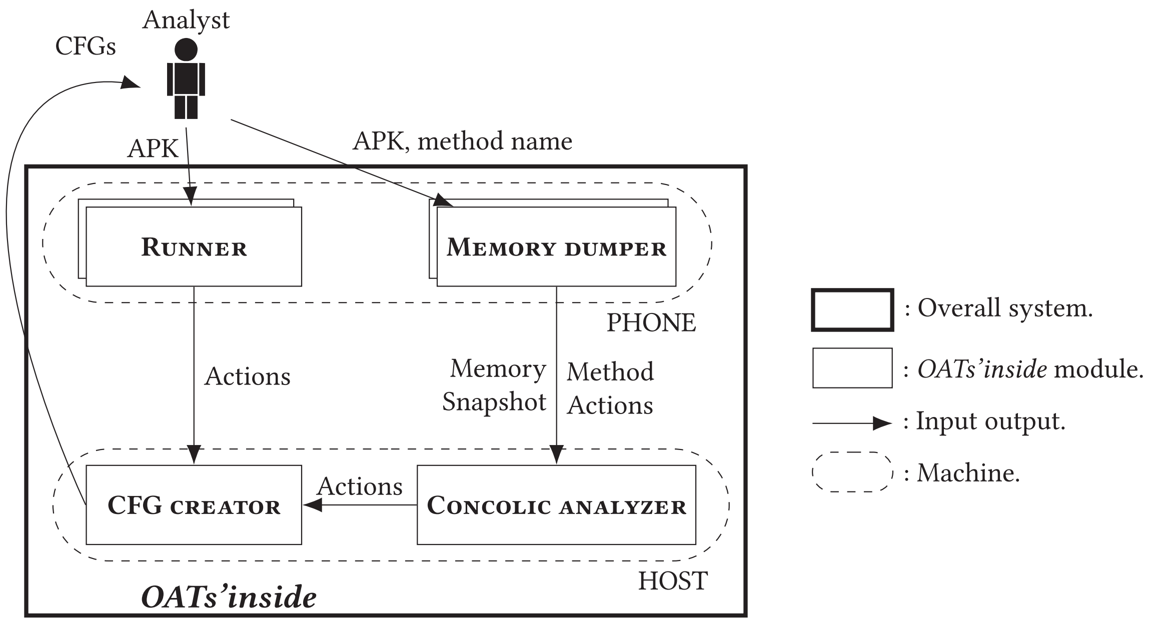


Fig. 1. *OATs'inside* architecture.

has three possible behaviors. It returns true when the `pin` is the correct one (1337) and false otherwise. `SimpleTestPIN.test` is obfuscated using the AOTC-based bytecode hiding: The bytecode is compiled into assembly and then removed, and the assembly is obfuscated by manually adding opaque predicates.

The whole analysis is driven by a human analyst who runs the application twice with two different PINs: a negative (-42), which generates an exception, and a wrong positive (42). The final objective of *OATs'inside* is to compute a CFG that best approximates the complete CFG, which is, for this example, given in Figure 2.

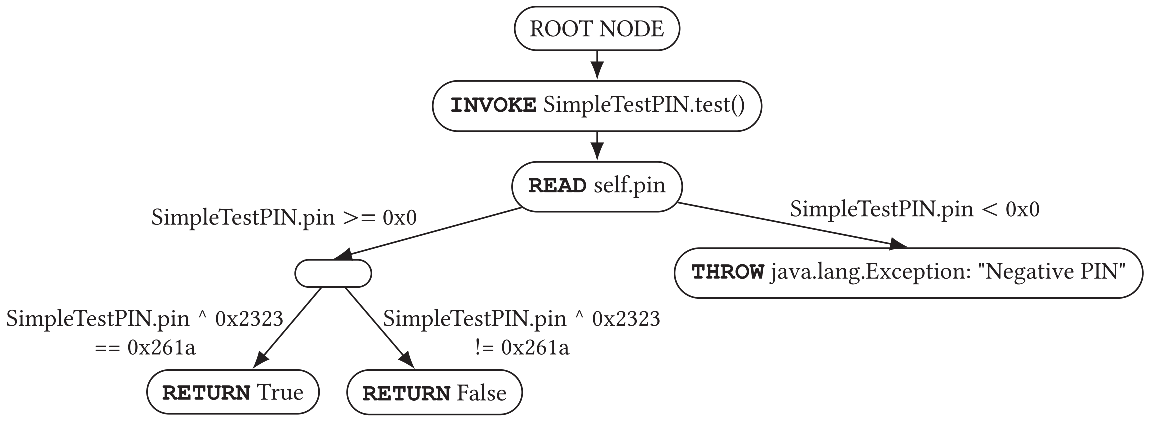
4.1 Runner Module

The RUNNER module is in charge of running the analyzed application and logging every object-level action performed by the application. There are nine different object-level actions [3]: invoking or returning a method, reading or writing an object field, allocating an object, entering or exiting a monitor session, and throwing or catching exceptions.

```java
// Two executions: test() with pin = -42 and pin = 42
public class SimpleTestPIN {
 public int pin = 0;
 public boolean test() throws Exception {
  if(this.pin < 0) throw new Exception("Negative PIN");
  if((this.pin ^ 0x2323) == 9754) // 1337^0x2323=9754
   return true;
  else return false;
}}
```

Listing 1. Simplified PIN test.

Applications contain three types of code: DEX, OAT, or native and *OATs'inside* should handle carefully their interactions. Indeed, state-of-the-art approaches suffer from one or more of the following limitations: They do



ROOT NODE
INVOKE SimpleTestPIN.test()
READ self.pin
SimpleTestPIN.pin >= 0x0
SimpleTestPIN.pin < 0x0
THROW java.lang.Exception: "Negative PIN"
SimpleTestPIN.pin ^ 0x2323 == 0x261a
SimpleTestPIN.pin ^ 0x2323 != 0x261a
RETURN True
RETURN False

Fig. 2. Expected output for `SimpleTestPIN.test`.

Table 2. Monitoring of Object Actions for Different Types of Executed Code

Analyzed binary	Invoke/return		Field access (read/write)		Object allocation/monitor		Expectation (throw/catch)	
	Method	Event type	Method	Event type	Method	Event type	Method	Event type
DEX	interpreter	direct	interpreter	direct	allocator	direct	exception handler	direct
OAT	class linker	breakpoint	disable heap	segv	allocator	direct	exception handler	direct
Native	class linker	breakpoint	disable heap or JNI	segv or direct	allocator	direct	exception handler	direct

not support OAT, arguing that the DEX bytecode is always available [5, 32, 34]; they do not collect all the possible actions [24, 32, 37, 38]; and they are bypassed by AOTC-based bytecode hiding, because they rely on JNI [32, 37]. The Runner module lifts these limitations by using monitor methods inside the ART library when possible and low-level debug methods otherwise.

Table 2 summarizes how each action is monitored, depending on the binary code type. If the action goes through the ART (all actions in the Dalvik bytecode, and object allocation, monitoring of the entry or exit, and exception handling in all code types), then a direct event is generated by adding a call to the logger inside the runtime. Otherwise, the action is retrieved by generating a low-level event based on debugging or memory protection capabilities. In particular, an OAT code that accesses (read or write) an object field is captured by disabling the heap memory: All heap accesses will generate a `segv` event. The same applies to native code when bypassing the JNI interface. Additionally, an OAT or a native code that invokes a method without calling the runtime is captured by hooking the address table and generating a breakpoint event.

Consequently, three types of events are generated or captured:

(1) Direct events: allocating an object, entering/exiting a monitor session, and throwing/catching an exception;
(2) Breakpoint events: invoking and returning a method;
(3) Segv events: reading or writing an object field.

To manage these events, we built the Runner module, a patch of the Android runtime whose main components are represented in Figure 3, where the three types of events are annotated as (D*N*), (B*N*), and (S*N*), respectively, where *N* designates in which order the components are chained. The runtime has information about high-level structures such as classes, signatures, and objects and also knows low-level entities such as register values, heap addresses, and kernel signals. Thus, patching the runtime allows us to bridge the semantic gap between the assembly and the bytecode world. The patch is divided into two entities: the `ProbeManager` and the `SignalManager`. The `ProbeManager` handles high-level events. It is the interface between the runtime and

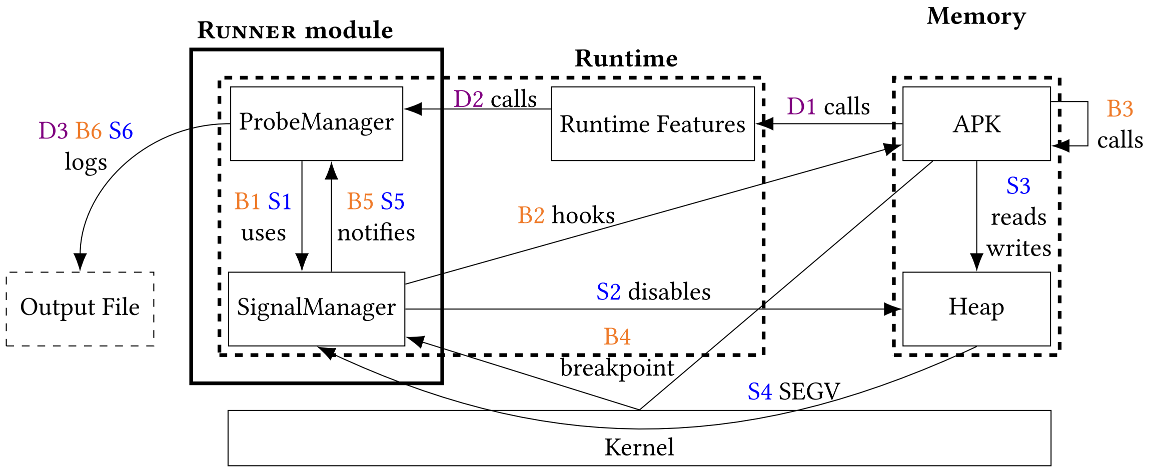


Fig. 3. Runtime patch architecture.

the output file when actions are logged. It logs object-level actions when they occur. The `SignalManager` handles low-level events. It sets breakpoints, handles kernel signals, and notifies the `ProbeManager` to log associated actions.

The `ProbeManager` logs each event with its associated instruction address. This allows linking of the bytecode events to the assembly code and will be used by the Concolic analyzer module (cf. Section 4.4). The thread identifier from which the event originates is also logged to avoid concurrent execution issues. In the following, we detail how the three types of events are handled by the Runner module.

Direct events. These events, indicated as "direct" in Table 2, are generated by the runtime library code. For example, when an object is allocated, the runtime allocator is called. The allocator allocates memory and returns it to the application. A call to the `ProbeManager`, containing the class of the allocated object, is added to the allocator before returning to the APK code. This part of the Runner module links the assembly world (the allocated address) and the bytecode world (the object class, independently of the executed code type). Entering or exiting a monitor session and throwing or catching an exception are logged using similar mechanisms in the runtime monitor and exception handler.

Breakpoint events. These events correspond to invoking or returning from a method. To log the invoke action, the Runner module needs to be notified when the first instruction of the method is executed. The classical way to do this would be to set a breakpoint at this address, catch the breakpoint (`SIGTRAP` signal), log the action, remove the breakpoint, and resume the execution. However, removing the breakpoint would prevent catching of future calls to this method. At first glance, instead of directly resuming the execution, a possible improvement would be to step one instruction, reset the breakpoint, and resume the execution. In this way, the breakpoint would be available for further calls. However, removing and then resetting the breakpoint would generate a concurrency issue if multiple threads execute the same method. In practice, every application runs more than five threads (GC, intents, profiler, etc.).

To solve this problem, the Runner module modifies the address of all the methods linked by the class linker in the runtime. As shown in Figure 4, the real address of the method (`code pointer`) is replaced by the address of a new dedicated area (`hooking area`). It contains a breakpoint instruction for generating the invoke action, the original address of the method's code (`original pointer`), and a pointer to the runtime internal structure representing the method (`class pointer`).

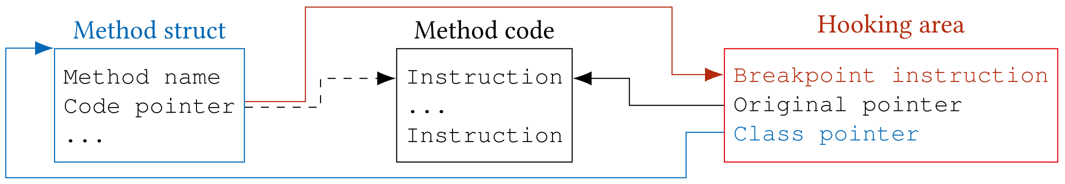


Fig. 4. Method invocation hooking process.

The same problem exists for catching the return event and is thus solved similarly. When this type of event occurs, as the breakpoint only carries the information about the executed address, we retrieve the method signature by using the runtime internal structure of the dedicated areas.

Segv events. These events correspond to accesses, by reading or writing, to object fields stored on the heap. Catching such accesses requires watching every load or store instruction to detect those that target heap addresses. To this end, the `SignalManager` uses the system page protection mechanism: It forbids all accesses to the heap memory pages using `mprotect`, causing any access to object fields to generate a fault, a `SEGV` kernel signal, which is caught by the `SignalManager`, which retrieves the faulty address. Then, the GC's internal structures are leveraged to map the assembly address to an object field. This is then transmitted to the `ProbeManager` for logging purposes. Finally, for an application to run as expected, heap access should actually be performed. The heap is re-enabled, and a single instruction is executed before disabling the heap again.

Running example output. Listing 2 gives the actions outputted by the Runner module for the running example, from lines 1 to 12 for the first execution and lines 13 to 18 for the second one. When these logs and the source code of Listing **??** are compared, it shows that most of the elements are retrieved. The access to the `pin` field (lines 3 and 4 and lines 15 and 16) is present for each execution of the method. The `throw` is divided into four events: the string creation (lines 5 and 6), the initialization of the exception object and its associated return (lines 7 and 8, and lines 9 and 10), and the throw itself (lines 11 and 12). Finally, the `return false` (lines 17 and 18) is detected. However, some Java actions are missing. The `return true`, which is never executed in our case, is not logged. The conditions are also lacking, as well as the usage of the allocated string (lines 5 and 6), which is a dependency of the `init` call (lines 7 and 8). This information is obtained by the remaining modules.

4.2 CFG Creator Module

The CFG creator module is in charge of creating the CFG. In fact, this graph is the union of the **interprocedural call graph (iCFG)** and the methods' olCFGs. These CFGs are built sequentially, using the events outputted by the Runner module: First, actions are split by method and the iCFG is created, and then the olCFG of each method is computed.

iCFG computation. The logs are split by method. The boundary of a method is defined by two properties of the Dalvik bytecode [3]. First, each method begins with an invoke and ends with a return. Then, each return is preceded by its corresponding invoke action. An invoke action is not necessarily followed by a return: Methods may never return (e.g., the main loops of graphical engines are infinite loops). Second, there is no jump across method bodies ("`goto`"-like statement). Thus, if a method m_b is invoked after a method m_a, then the return of m_a cannot occur before the return of m_b. Invocations cannot be interleaved. Thanks to this last remark, we can split actions by method by reconstructing the call stack. During the call stack computation, the iCFG is made: When an invoke event occurs, an edge is added between it and the last method. Note that actions are mixed between different threads. The inclusion of the thread identifier in logs allows each thread CFG to be built independently.

olCFG computation. We define the olCFG of a method the graph whose nodes contain Java-level actions (as listed later in Table 2), and whose edges represent the execution traces. Each node contains, for each execution:

```
tid: 3520, event_address: 512236427828
invoke SimpleTestPIN;test()
tid: 3520, event_address: 512236429712
read SimpleTestPIN;pin => -42
tid: 3520, event_address: 512236429884
newObj String => 315654920
tid: 3520, event_address: 512236429832
invoke java/lang/Exception;<init>((String) 315654920)
tid: 3520, event_address: 512236429836
return void
tid: 3520, event_address: 512236429844
throw java.lang.Exception("Negative PIN")
tid: 3520, event_address: 512236427908
invoke SimpleTestPIN;test()
tid: 3520, event_address: 512236429712
read SimpleTestPIN;pin = > 42
tid: 3520, event_address: 512236427912
return false
```

Listing 2. RUNNER module output on `SimpleTestPIN`.

- the address of the instruction that operated on the Java object: the address of the instruction in the case of native code; the index of the instruction in the bytecode, otherwise;
- the execution number, to distinguish between several executions of the application;
- the nature of the operation (INVOKE, READ, etc.) as listed in Table 2;
- parameters of this operation (name of the called function, read field, address of the object of interest, etc.);
- values returned by the considered operation.

The olCFG computation requires as input the sequence of actions of a method. Each node is uniquely characterized by the address of the assembly instruction generating the action. Thus, if the same address is executed multiple times (several executions of the method or loops in the method's body), then the node representing this action contains the details of all executed actions. For example, in Figure 5, the node `7743aba190` contains two read actions from two different executions: The first read obtains the value `-42`, and the second one `42`. A code performing a long loop could flood the log with a repeated action about this node. We have optimized the CFG creator module to reduce this problem that also occurs for multiple executions of the same method: When an execution does not add new actions to the olCFG, the execution log is not translated into the graph. We could, as a future work, perform the same optimization in real time for a single execution.

A special root node is added to mark the beginning of the method. When iterating over the sequence of actions, the algorithm creates an edge from the current instruction to the next one. When a node holds several actions, several destinations can follow, hence revealing the existence of a condition whose nature is not known yet. Note that if ASLR is activated, then addresses change between two different executions, breaking the node unicity previously mentioned. Thus, for our experiments ASLR have been disabled. Nevertheless, we believe that using offsets to the base address of the loaded binary would solve this problem.

Running example output. Figure 5 shows the olCFG computed for the `SimpleTestPIN.test` method. This is a human-readable representation of Listing 2. The same elements are missing: The "return true" case is not present, the condition expressions are missing, and the dependency between the allocation and the invocation is not explicit. Thus, the analyst cannot retrieve the correct PIN number: He/she cannot identify the conditions that need to be satisfied, because they are not present.

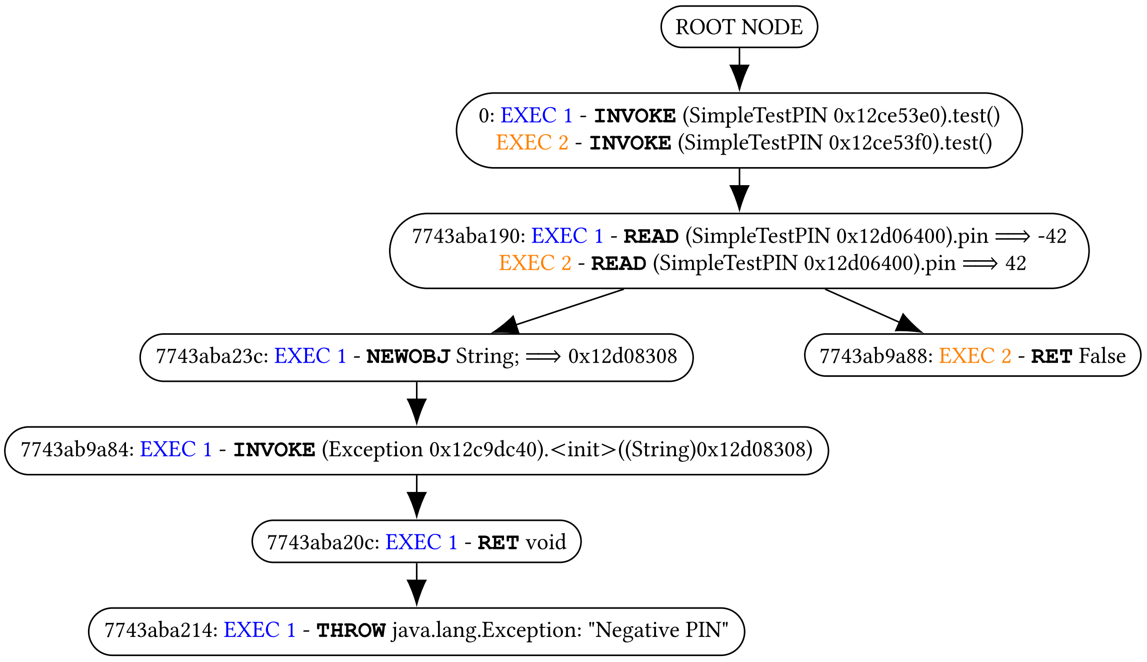

ROOT NODE
0: EXEC 1 - INVOKE (SimpleTestPIN 0x12ce53e0).test()
EXEC 2 - INVOKE (SimpleTestPIN 0x12ce53f0).test()
7743aba190: EXEC 1 - READ (SimpleTestPIN 0x12d06400).pin ⟹ -42
EXEC 2 - READ (SimpleTestPIN 0x12d06400).pin ⟹ 42
7743aba23c: EXEC 1 - NEWOBJ String; ⟹ 0x12d08308
7743ab9a88: EXEC 2 - RET False
7743ab9a84: EXEC 1 - INVOKE (Exception 0x12c9dc40).<init>((String)0x12d08308)
7743aba20c: EXEC 1 - RET void
7743aba214: EXEC 1 - THROW java.lang.Exception: "Negative PIN"

Fig. 5. oICFG of `SimpleTestPIN.test`.

4.3 Memory Dumper Module

The Memory dumper module is responsible for making snapshots of the memory. This module is called just before the execution of a method and dumps the whole memory or the process. These snapshots give the method's code and data to the Concolic analyzer module in charge of the symbolic execution. In this way, if a method is used as a place holder for several unpacked assembly codes, then each snapshot will provide the current version of the code. This module can be either activated by the Runner at each method execution (but it considerably slows down the analysis) or activated on demand by the human analyst.

4.4 Concolic Analyzer Module

The Concolic analyzer module executes symbolically the dumped assembly code and uses the values observed when actions occurred to help the symbolic execution. The first step allows us to build a CFG describing the execution paths explored during the dynamic analysis; however, it lacks both conditional expressions and how variables are manipulated by the actions. Such knowledge is important for the analyst, because it helps to understand the behavior execution. For example, in Figure 5, the parameter (`12d08308`) of the invoke (`7743ab9a84`) should be linked with the preceding allocation.

The Concolic analyzer module takes as input the list of actions logged by the Runner module and all the memory snapshots made by the Memory Dumper module. It generates the conditional expressions at branching nodes and the data dependencies between variables. This is done in three steps:

(1) the assembly is annotated with breakpoints at action addresses;
(2) the symbolic execution is started until it reaches a breakpoint or a condition;
(3) if the symbolic analysis is stopped, then the module respectively logs the corresponding action or condition for
   (a) a breakpoint;
   (b) a condition.

Assembly breakpoints. In the assembly code returned by the Memory Dumper module, a breakpoint is set for all generated actions. For example, we set a breakpoint at the address `7743aba190` (READ `pin` field action), which corresponds to the instruction `ldr w2, [x1, #12]`.

Symbolic execution. The symbolic execution is initialized: The `PC` is set to the entry point of the method and a symbolic value is created for each method parameter. The symbolic execution can stop for one of three reasons: a breakpoint, a condition, or the end of the method is reached.

Analysis stop and concretization. When the symbolic execution is stopped, the analysis flow is guided and symbolic values are managed. Two types of stops are handled:

(1) Breakpoint: First, if the action type is allocation, read, or return, then a new symbolic value is created, named according to the Java class or field name. For example, the read at address `7743aba190` creates a symbol "SimpleTestPIN.pin," as shown in Listing 3, line 5. The instruction output register `w2` is set to this new symbolic value. Second, if the action type is read, write, or invoke, then the read register or memory value is retrieved. If this expression is symbolic, then it is outputted. For example, the parameter of the invoke is logged in line 13 with the name created previously in line 10.
(2) Condition: The symbolic engine provides the two symbolic conditions corresponding to the two branches. They are concretized: Symbolic values are replaced by the concrete value given by the trace. The condition that holds is logged and the symbolic execution is resumed, taking the corresponding path. Note that the concretization happens only for logging and choosing the branch: Registers and memory stay symbolic to continue tracking data dependencies.

One key point of this symbolic analysis is that no SMT solver is ever called. Instead, only value replacement (concretization) is made. Moreover, the analysis always follows only one path. This saves the analysis from the usual drawbacks of symbolic analysis that could lead to high execution time or memory space overhead [6].

Finally, the results of the Concolic analyzer module are sent to the CFG creator module to improve the olCFG. Blank nodes are added, in two cases:

- a node that has not (yet) been explored: These nodes do not have outgoing edges. These nodes are needed, because a unique execution will not cover all possible execution paths. This is up to the analyst to choose whether he wants to explore this path.
- a node without any observable Java-level action: These nodes have outgoing edges. They are used to preserve the structure of the olCFG to reflect that multiple paths can be taken from this node.

This is the final human-readable output of *OATs'inside*.

Running example output. After the Concolic analyzer execution on the snapshot and the actions retrieved for the running example described in Section 4.1, the enriched list of actions is given in Listing 3. Compared to Listing 2, three new condition events have been added. The first two (lines 6 and 7 and lines 23 and 24) are the opposite, because they represent the same condition that is taken or not. It corresponds to the check that the `pin` field is positive, which is expressed in the condition expression written in the listing. The third condition (lines 25 and 26) is the comparison to the correct pin value, which is XORed. The symbol `SimpletestPIN.pin` has been concretized by 42 using line 21 to choose the branch to execute. Moreover, four symbolic annotations have been added. The two lines attached to the read (lines 5 and 22) and the line attached to the allocation (line 10) represent the new symbolic values created. The remaining one (line 13) shows that the symbol representing the output of the allocation (line 10) is directly used as an invocation parameter when calling the constructor exception `<init>`. The updated olCFG of the `test` method is shown in Figure 6. An analyst can now easily understand how the PIN is handled.

```
tid: 3520, event_address: 512236427828
invoke SimpleTestPIN;test()

tid: 3520, event_address: 512236429712
read SimpleTestPIN;pin => -42
symb: "SimpleTestPIN.pin"

tid: 3520, event_address: 512236429716
condition "(LShR(SimpleTestPIN.pin, 0x1f) & 0x1) != 0x0"

tid: 3520, event_address: 512236429884
newObj String => 315654920
symb: "new_ui64"

tid: 3520, event_address: 512236429832
invoke java/lang/Exception;<init>((String) 315654920)
symb: ["new_ui64"]

tid: 3520, event_address: 512236429836
return void

tid: 3520, event_address: 512236429844
throw java.lang.Exception("Negative PIN")

tid: 3520, event_address: 512236427908
invoke SimpleTestPIN;test()

tid: 3520, event_address: 512236429712
read SimpleTestPIN;pin => 42
symb: "SimpleTestPIN.pin"

tid: 3520, event_address: 512236429716
condition "(LShR(SimpleTestPIN.pin, 0x1f) & 0x1) == 0x0"

tid: 3520, event_address: 512236429736
condition "(SimpleTestPIN.pin ^ 0x2323) != 0x261a"

tid: 3520, event_address: 512236427912
return false
```

Listing 3. Concolic analyzer output on `SimpleTestPIN`.

5 IMPLEMENTATION DETAILS

The full implementation is available at https://gitlab.inria.fr/cidre-public/oatinside. The Runner and Memory dumper modules are written in C++, inside the original ART. The architecture-specific parts have been developed only for ARMv8 [4]: signal handling, the hooking process presented in Figure 4, and specific instructions handling mutual exclusion that *OATs'inside* may interrupt (`ldx*` and `stx*`). ARMv7 could be supported, and other architectures are marginal. The communication between the phone and the host is established using protobuf[4] over a socket connection. The CFG creator module is based on the NetworkX Python library [16]. The Concolic analyzer module uses angr [28] as a symbolic execution engine.

A few optimization techniques have been implemented for the Runner and Memory dumper modules. First, system libraries and applications are whitelisted. Indeed, when one of their methods is invoked, the heap is re-enabled, improving the execution time. Second, we cache the mapping between addresses and object fields used by the Runner module. This cache is flushed when the GC is triggered, because it may move objects around.

[4]Protobuf: https://developers.google.com/protocol-buffers/.

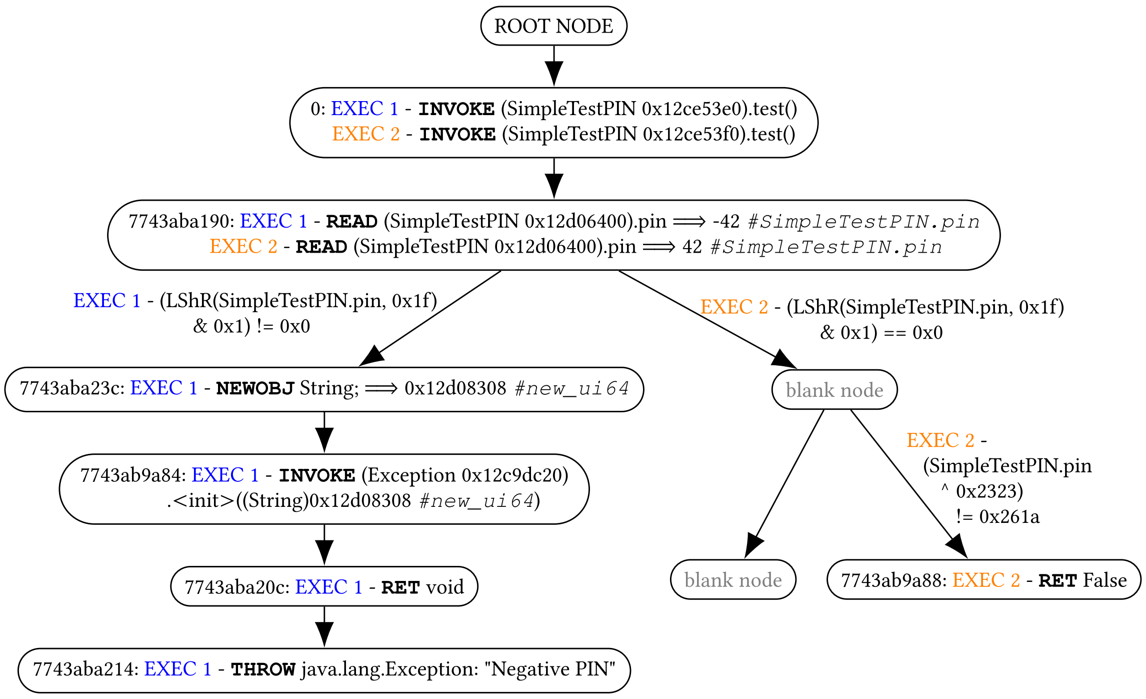

ROOT NODE
0: EXEC 1 - INVOKE (SimpleTestPIN 0x12ce53e0).test()
EXEC 2 - INVOKE (SimpleTestPIN 0x12ce53f0).test()
7743aba190: EXEC 1 - READ (SimpleTestPIN 0x12d06400).pin ⟹ -42 #SimpleTestPIN.pin
EXEC 2 - READ (SimpleTestPIN 0x12d06400).pin ⟹ 42 #SimpleTestPIN.pin
EXEC 1 - (LShR(SimpleTestPIN.pin, 0x1f) & 0x1) != 0x0
EXEC 2 - (LShR(SimpleTestPIN.pin, 0x1f) & 0x1) == 0x0
7743aba23c: EXEC 1 - NEWOBJ String; ⟹ 0x12d08308 #new_ui64
blank node
7743ab9a84: EXEC 1 - INVOKE (Exception 0x12c9dc20) .<init>((String)0x12d08308 #new_ui64)
EXEC 2 - (SimpleTestPIN.pin ^ 0x2323) != 0x261a
7743aba20c: EXEC 1 - RET void
blank node
7743ab9a88: EXEC 2 - RET False
7743aba214: EXEC 1 - THROW java.lang.Exception: "Negative PIN"

Fig. 6. olCFG of `SimpleTestPIN.test`.

The Runner and Memory dumper modules need a few kernel changes. First, when the Runner module handles `SEGV` events, it has to authorize accesses to the heap and to execute a single instruction before disabling the heap again. To avoid concurrent accesses to the heap in the meanwhile, a thread-oriented `mprotect` has been added to the kernel [25]. The Memory dumper also needs a mechanism to pause the execution of threads, which is not available in the Linux kernel. A kernel syscall fulfilling that task has been added, using the freezing mechanism that the kernel already uses for hibernation (suspend to disk) [35].

These modules also highly rely on two signal handlers set up for the `SIGTRAP` and `SEGV` signals. To prevent them from being replaced or removed by the application, we added a new syscall. This syscall sets up definitive signal handlers whose addresses are given in the parameter. The sigaction kernel syscall is modified so that this handler can never be modified. If the signal is generated by *OATs'inside*, it is treated; if not, then it is forwarded to the application installed handler. This implementation solves practical problems due to library helpers for native development such as Google Breakpad[5] or **Application Crash Reports for Android (ACRA)**.[6]

Finally, the implementation of the Runner and the Memory dumper modules needs to differentiate segmentation faults (`SEGV`) that are raised by read or write operations. However, for ARMv8, no distinction is made in the kernel. As a first approximation, the distinction is made by comparing the accessed value before and after the execution. If they are the same, then the module considers it as a read, else as a write.

6 EVALUATION AND DISCUSSION

We evaluated *OATs'inside* to answer the five following questions:

[5]Google Breakpad: https://github.com/google/breakpad.
[6]ACRA: https://github.com/ACRA/acra.

Table 3. *OATs'inside* Compared with State-of-the-art Solutions

	Original source code	**DEX**															**Native**														
	Analyzed binary	DEX only					Pack DEX					OAT only					JNI					JNI+obf					DHA				
	Evaluated tool	**O**	A	T	J	M	**O**	A	T	J	M	**O**	A	T	J	M	**O**	A	T	J	M	**O**	A	T	J	M	**O**	A	T	J	M
Method	Invoke / Return	Ⓡ	●	●	●	●	Ⓡ	-	●	-	○	Ⓡ	-	-	-	○	Ⓡ	-	-	●	○	Ⓡ	-	-	●	○	▨	▨	▨	▨	▨
Allocation	Object	Ⓡ	●	●	●	●	Ⓡ	-	●	-	-	Ⓡ	-	-	-	-	Ⓡ	-	-	-	-	Ⓡ	-	-	-	-	▨	▨	▨	▨	▨
	Primitive variable	Ⓡ	●	●	●	●	Ⓡ	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	▨	▨	▨	▨	▨
	Primitive Array	Ⓡ	●	●	●	●	Ⓡ	-	●	-	-	Ⓡ	-	-	-	-	Ⓡ	-	-	-	-	Ⓡ	-	-	-	-	▨	▨	▨	▨	▨
Access	Object Field	Ⓡ	●	●	●	●	Ⓡ	-	●	-	○	Ⓡ	-	-	-	○	Ⓡ	-	-	●	○	Ⓡ	-	-	●	○	Ⓡ	-	-	-	-
	Primitive variable	Ⓡ	●	●	●	●	Ⓡ	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	▨	▨	▨	▨	▨
	Primitive Array	Ⓡ	●	●	●	●	Ⓡ	-	●	-	○	Ⓡ	-	-	-	○	Ⓡ	-	-	●	○	Ⓡ	-	-	●	○	Ⓡ	-	-	-	-
Operations	Object Field	Ⓡ	●	●	●	●	Ⓡ	-	●	-	○	Ⓡ+Ⓒ	-	-	-	○	Ⓡ+Ⓒ	-	-	○	○	Ⓡ+Ⓒ	-	-	-	○	Ⓡ+Ⓒ	-	-	-	-
	Primitive variable	Ⓡ	●	●	●	●	Ⓡ	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	▨	▨	▨	▨	▨
	Primitive Array	Ⓡ	●	●	●	●	Ⓡ	-	●	-	○	Ⓡ+Ⓒ	-	-	-	○	Ⓡ+Ⓒ	-	-	○	○	Ⓡ+Ⓒ	-	-	-	○	Ⓡ+Ⓒ	-	-	-	-
Condition	Object Field	Ⓡ	●	●	●	●	Ⓡ	-	●	-	○	Ⓡ+Ⓒ	-	-	-	○	Ⓡ+Ⓒ	-	-	○	○	Ⓡ+Ⓒ	-	-	-	○	Ⓡ+Ⓒ	-	-	-	-
	Primitive variable	Ⓡ	●	●	●	●	Ⓡ	-	●	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	▨	▨	▨	▨	▨
	Primitive Array	Ⓡ	●	●	●	●	Ⓡ	-	●	-	○	Ⓡ+Ⓒ	-	-	-	○	Ⓡ+Ⓒ	-	-	○	○	Ⓡ+Ⓒ	-	-	-	○	Ⓡ+Ⓒ	-	-	-	-
Typing	Check	Ⓡ	●	●	●	●	Ⓡ	-	●	-	-	?	-	-	-	-	Ⓡ	-	-	-	-	Ⓡ	-	-	-	-	▨	▨	▨	▨	▨
	Cast	Ⓡ	●	●	●	●	Ⓡ	-	●	-	-	?	-	-	-	-	?	-	-	-	-	?	-	-	-	-	▨	▨	▨	▨	▨
Exception	Throw / Catch	Ⓡ	●	●	●	●	Ⓡ	-	●	-	-	Ⓡ	-	-	-	-	Ⓡ	-	-	-	-	Ⓡ	-	-	-	-	▨	▨	▨	▨	▨
Monitor	Enter / Exit	Ⓡ	●	●	●	●	Ⓡ	-	●	-	-	Ⓡ	-	-	-	-	Ⓡ	-	-	-	-	Ⓡ	-	-	-	-	▨	▨	▨	▨	▨

O: *OATs'inside*; T: TIRO [34]; A: ARTist [5]; J: JN-SAF [32]; M: Malton [37]
●: fully, ○: partially
Ⓡ: Retrieval requires the Runner module
Ⓡ+Ⓒ: Retrieval requires the Concolic analyzer module
?: Retrieval requires more static analyses

Q1: *Can OATs'inside retrieve the object behavior of an application regardless of the use of obfuscation techniques?*
Q2: *What is the contribution of OATs'inside to the state-of-the-art tools?*
Q3: *Is OATs'inside practical to use?*
Q4: *What is the overhead of OATs'inside?*
Q5: *Is OATs'inside stealth and robust when analyzing malware?*

6.1 Robustness against Obfuscation

To assess the proper functioning of *OATs'inside*, we designed unit tests, as reported in Table 3. To build the test cases, we read the Dalvik bytecode specification [3] to enumerate all possible Java source statement behaviors (allocations, register and memory accesses, arithmetic and bitfield operations, condition checks, type checks, exception management, critical session monitoring). These behaviors are divided into eight families and 20 categories listed in the two first columns of Table 3. For each behavior categories, we distinguish, when relevant, three Java types: object, primitive variable or primitive array. For example, the condition category represents changing the execution flow (`if` statement), depending on the value of an object field, a primitive variable allocated onto the stack, or primitive array element. Finally, we obtain 70 unit test classes for which we check the output log of *OATs'inside*.

To handle the different obfuscation techniques, the test cases were packaged in a single application obfuscated in six different versions:

(1) the *DEX only* version corresponding to test cases made in Dalvik bytecode;
(2) the *Pack DEX* version resulting from the usage of a bytecode native packer on the *DEX only* version;
(3) the *AOTC* version produced by using the AOTC-based bytecode hiding technique (cf. Section 2.2);

(4) the *JNI* version corresponding to test cases implemented in C++ using JNI;
(5) the *JNI+obf* version resulting from the usage of the Tigress obfuscator [10]:
  - Recipe #1: Virtualize [7]: This recipe intends to break the control flow by adding a level of indirection (a virtual machine) into the code;
  - Recipe #2: Encode Arithmetic [7]: This recipe intends to complexify the arithmetic calculus by introducing equivalent larger formulas.
(6) the *DHA* version resulting from the usage of the DHA methods (cf. Section 2.2) on the *JNI* version. DHA method obfuscates only heap accesses (others behaviors are grayed in Table 3).

When running the test cases, we manually checked that the olCFG outputted by *OATs'inside* corresponded to the expected one. We indicated in Table 3 whether the test case evaluation required only the Runner module (indicated as Ⓡ) or both the Runner module and the Concolic analyzer module (indicated as Ⓡ+Ⓒ).

Results show that *OATs'inside* retrieves almost every behavior when no obfuscation is used except for two cases. First, as other tools, *OATs'inside* does not retrieve behavior based on primitive variables. This is expected, because variables are allocated onto the stack that is not monitored. Missing these variable-oriented behaviors for native code is not an important limitation, because they are still considered by the symbolic execution. Second, *OATs'inside* does not retrieve bytecode behaviors that are removed at compilation time (e.g., type checking). However, this could be retrieved using more advanced static analysis such as type propagation and checking [9].

For the obfuscated version of the unit tests, *OATs'inside* captures partially the Java behaviors. All operations are observed with recipe #1. Recipe #1 uses a Virtualize Machine (of a custom bytecode) to hide the real control flow of the code of our functions. In this case, the symbolic execution is still able to track values and all types of Java operations are observed: The control flow introduced by the VM does not depend on the Java values and thus the symbolic analysis does not output this new flow in the olCFG. With recipe #2, arithmetic expressions are replaced by more complex ones. *OATs'inside* manages to retrieve the correct operations thanks to angr that is able to simplify them. Nevertheless, we think that applying recipe #2 on more complex codes would have fooled *OATs'inside*, which may not be able to simplify the expressions (angr solves simple ones but would be limited on complex ones).

6.2 *OATs'inside* Functional Contribution

To assess the contribution of *OATs'inside* to the state of the art, we minutely read the papers describing four tools: TIRO [34], ARTist [5], JN-SAF [32], and Malton [37]. In Table 3, each reported column corresponds to a tool, *OATs'inside* being the first column.

First, because the DHA obfuscation is a new technique, none of the other tools passed the test cases of the *DHA* version. In fact, tools that focus on tracing accesses to object fields all rely on a clearly defined JNI interface.

ARTist [5], as mentioned in Section 3, does not target native code but only bytecode. This explains why it only retrieves behaviors for the *DEX only* version.

TIRO [34] is an unpacker. Thus, it can output the loaded bytecode of the *Pack DEX* version. Then, the bytecode being available, the analyst can retrieve all the application behaviors. However, excluding the behavior related to code loading, which is not an elementary Java behavior, TIRO does not analyze any native code.

JN-SAF [32] and Malton [37] both do taint flow analysis. Therefore, they do not care about allocations, typing, exceptions, or monitoring of events and do not output them at all.

JN-SAF [32] is a static analysis-based taint tracking tool. Owing to its static nature, it cannot work with the *Pack DEX* version. Moreover, JN-SAF targets only native methods and does not handle AOTC-compiled code and, thus, it misses the *AOTC* version. Because JN-SAF aims at tracking flows, it does not log explicitly the conditions and the operations made by the code. However, these elements are part of the flow process, i.e.„ the propagation algorithm. That is why some partial information about operations and conditions is captured. Finally, because JN-SAF relies on classical symbolic execution, obfuscated assembly can overload its analysis, preventing the *JNI+obf* version from being handled.

Malton [37] is a hybrid tool that realizes data taint tracking over framework libraries and system calls. Therefore, it does not retrieve information about the internal code methods and classes. That is why all its outputs are qualified as partial. Moreover, because Malton is dynamic, it can handle the *Pack DEX* version. The symbolic analysis that is conducted is concolic: Like *OATs'inside*, it follows the execution and, thus, is not sensitive to obfuscation (unlike JN-SAF) and can tackle the *JNI+obf* version. Finally, Malton works with the *AOTC* version, because it does not rely on the APK structure but bases all its analysis on executed assembly instructions.

Thanks to the conducted review of state-of-the-art tools, we believe that *OATs'inside* is the only one that can work with DHA obfuscated applications and recover a CFG of the executed part of the code for every tested categories.

6.3 Analysis of OWASP UnCrackable App for Android

To assess if *OATs'inside* is practical to use, then we used *OATs'inside* to reverse an application dubbed *UnCrackable App for Android*. This application was developed by OWASP, a nonprofit foundation that works on improving the security of software. To that end, they released a manual called **"Mobile Security Testing Guide" (MSTG)**[7] that aims at defining the industry standard for mobile application security. *UnCrackable* apps[8] are examples used among this manual. Four levels are proposed. The first level is a pure Java application hiding a password: It is of no interest for *OATs'inside*. The next three levels uses native code to evaluate the password entered by the users. In this section, we present in detail how *OATs'inside* solves the level 2, and we briefly explain levels 3 and 4.

6.3.1 Level 2. UnCrackable level 2 is presented as an application that "hides away data and functionality in native libraries"[9] and tries to fool debuggers. The goal for a reverser is to find a password that makes the application show "Success." This is a well-suited target for evaluating *OATs'inside*'s capability against real-world applications, since it is precisely designed to use native code to complexify analyses and to represent what a real developer could do.

First observations. First, to analyse this application, we installed it on a real phone and ran it using *OATs'inside*. After launching, the application shows a screen asking for a "secret string." By typing "password test" and clicking on the verify button, the application answers "That's not it. Try again." We then closed the application.

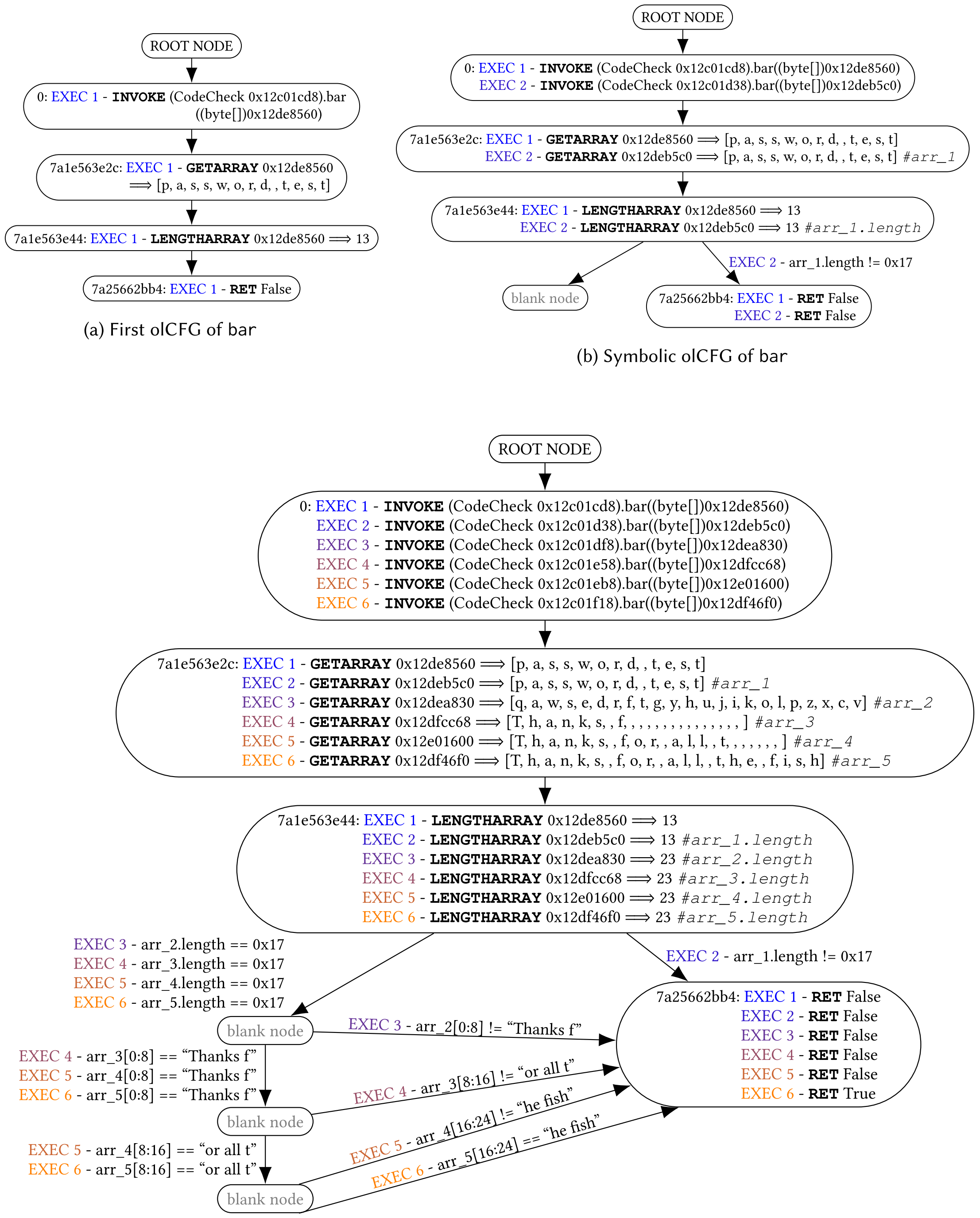
During this first analysis, *OATs'inside* traced 2,455 methods. Since the methods are grouped inside their respective classes and packages (around five hundreds), we started by generating the olCFG for all methods that do not belong to a default Android library (e.g., `java.*`, `android.*`). Then, we searched in the 15 remaining methods for anything that could be related to password checking. During this search, we first found three interesting methods:

- `b.a`: searches for the binary `su` using the `$PATH` environment variable.
- `b.b`: searches for the string "test-keys" in the build tags.
- `b.c`: searches for rooting applications artifacts such as `Superuser.apk` or `/system/etc/.has_su_deamon`.

All these anti-analysis tricks are transparently avoided, since *OATs'inside* does not need to root Android or to use an emulator but rather comes as a genuine Android version.

Finally, we generated an olCFG where the string that we have entered ("password test") appears. This is the `bar` method's olCFG, shown in Figure 7(a). This method is a good candidate to deeper analysis: It returns a Boolean that could indicate a good or bad password and manipulates the entered password by computing its length.

[7] OWASP MSTG: https://mobile-security.gitbook.io/.
[8] *UnCrackable* levels 1 to 4 source code: https://github.com/OWASP/mastg-crackmes/tree/master/Android.
[9] *UnCrackable* level 2 presentation: https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering#bypassing-debugger-detection.



ROOT NODE
0: EXEC 1 - INVOKE (CodeCheck 0x12c01cd8).bar ((byte[])0x12de8560)
7a1e563e2c: EXEC 1 - GETARRAY 0x12de8560 ⟹ [p, a, s, s, w, o, r, d, , t, e, s, t]
7a1e563e44: EXEC 1 - LENGTHARRAY 0x12de8560 ⟹ 13
7a25662bb4: EXEC 1 - RET False
ROOT NODE
0: EXEC 1 - INVOKE (CodeCheck 0x12c01cd8).bar((byte[])0x12de8560)
EXEC 2 - INVOKE (CodeCheck 0x12c01d38).bar((byte[])0x12deb5c0)
7a1e563e2c: EXEC 1 - GETARRAY 0x12de8560 ⟹ [p, a, s, s, w, o, r, d, , t, e, s, t]
EXEC 2 - GETARRAY 0x12deb5c0 ⟹ [p, a, s, s, w, o, r, d, , t, e, s, t] #arr_1
7a1e563e44: EXEC 1 - LENGTHARRAY 0x12de8560 ⟹ 13
EXEC 2 - LENGTHARRAY 0x12deb5c0 ⟹ 13 #arr_1.length
EXEC 2 - arr_1.length != 0x17
blank node
7a25662bb4: EXEC 1 - RET False
EXEC 2 - RET False
ROOT NODE
0: EXEC 1 - INVOKE (CodeCheck 0x12c01cd8).bar((byte[])0x12de8560)
EXEC 2 - INVOKE (CodeCheck 0x12c01d38).bar((byte[])0x12deb5c0)
EXEC 3 - INVOKE (CodeCheck 0x12c01df8).bar((byte[])0x12dea830)
EXEC 4 - INVOKE (CodeCheck 0x12c01e58).bar((byte[])0x12dfcc68)
EXEC 5 - INVOKE (CodeCheck 0x12c01eb8).bar((byte[])0x12e01600)
EXEC 6 - INVOKE (CodeCheck 0x12c01f18).bar((byte[])0x12df46f0)
7a1e563e2c: EXEC 1 - GETARRAY 0x12de8560 ⟹ [p, a, s, s, w, o, r, d, , t, e, s, t]
EXEC 2 - GETARRAY 0x12deb5c0 ⟹ [p, a, s, s, w, o, r, d, , t, e, s, t] #arr_1
EXEC 3 - GETARRAY 0x12dea830 ⟹ [q, a, w, s, e, d, r, f, t, g, y, h, u, j, i, k, o, l, p, z, x, c, v] #arr_2
EXEC 4 - GETARRAY 0x12dfcc68 ⟹ [T, h, a, n, k, s, , f, , , , , , , , , , , , , , ,] #arr_3
EXEC 5 - GETARRAY 0x12e01600 ⟹ [T, h, a, n, k, s, , f, o, r, , a, l, l, , t, , , , , , ,] #arr_4
EXEC 6 - GETARRAY 0x12df46f0 ⟹ [T, h, a, n, k, s, , f, o, r, , a, l, l, , t, h, e, , f, i, s, h] #arr_5
7a1e563e44: EXEC 1 - LENGTHARRAY 0x12de8560 ⟹ 13
EXEC 2 - LENGTHARRAY 0x12deb5c0 ⟹ 13 #arr_1.length
EXEC 3 - LENGTHARRAY 0x12dea830 ⟹ 23 #arr_2.length
EXEC 4 - LENGTHARRAY 0x12dfcc68 ⟹ 23 #arr_3.length
EXEC 5 - LENGTHARRAY 0x12e01600 ⟹ 23 #arr_4.length
EXEC 6 - LENGTHARRAY 0x12df46f0 ⟹ 23 #arr_5.length
EXEC 3 - arr_2.length == 0x17
EXEC 4 - arr_3.length == 0x17
EXEC 5 - arr_4.length == 0x17
EXEC 6 - arr_5.length == 0x17
EXEC 2 - arr_1.length != 0x17
blank node
EXEC 3 - arr_2[0:8] != "Thanks f"
EXEC 4 - arr_3[0:8] == "Thanks f"
EXEC 5 - arr_4[0:8] == "Thanks f"
EXEC 6 - arr_5[0:8] == "Thanks f"
blank node
EXEC 4 - arr_3[8:16] != "or all t"
EXEC 5 - arr_4[8:16] == "or all t"
EXEC 6 - arr_5[8:16] == "or all t"
blank node
EXEC 5 - arr_4[16:24] != "he fish"
EXEC 6 - arr_5[16:24] == "he fish"
7a25662bb4: EXEC 1 - RET False
EXEC 2 - RET False
EXEC 3 - RET False
EXEC 4 - RET False
EXEC 5 - RET False
EXEC 6 - RET True

(a) First olCFG of bar

(b) Symbolic olCFG of bar

(c) Final olCFG of bar , Vol. 1, No. 1, Article . Publication date: February 2023.

Fig. 7. olCFGs of bar generated by *OATs'inside*.

Deeper analysis of the `bar` *method.* We chose to deeply analyse the `bar` method by leveraging the concolic analysis of *OATs'inside*. To use the Concolic analyzer, we re-ran the application entering the same inputs ("password test") using the Memory dumper. With the help of the output dump, *OATs'inside* generated the olCFG shown in Figure 7(b). Based on it, we can state that `False` is returned when the length of the entered password is different from 23 (0x17 in the graph). Thus, we re-ran the application but, this time, entering a 23 character-long password ("qawsedrftgyhujikolpzxcv"). The corresponding concolic-improved olCFG was computed and revealed that `False` is also returned if the eight first characters of the password are not "Thanks f." By repeating this process three other times, i.e., setting the password accordingly to the olCFG, we obtained the olCFG in Figure 7(c). In this olCFG, the correct password ("Thanks for all the fish") appears immediately in the sixth execution, which returns `True`. This password has been validated using the application.

Remarks. Using *OATs'inside*, we have been able to analyze transparently an obfuscated native Android application. Indeed, by looking at the *UnCrackable* source code[10]: We can note that some native anti-debugger tricks such as auto-debugging have not even been noticed during the analysis. While the analyst still has to guide the tool using its expertise, knowledge, and time for de-obfuscating the application are not necessary. Note that, even if all modules presented in Figure 1 communicate automatically, the analyst still has the opportunity to adapt his choices with the findings after an execution. Additionally, some other tools such as IntelliDroid [33] may give relevant information to assist its investigation. The easy retrieval of the secret password shows that *OATs'inside* is of practical use.

6.3.2 Level 3. OATs'inside helps to solve similarly the level 3 of the OWASP Uncrackable application. Using 25 executions, we have incrementally retrieved: (1) the length of the password and (2) the 24 characters of the password, one by one. The olCFG obtained has been given in Appendix A. When applying the method previously described, the only noticeable difference for level 3 is the number of comparisons made by the application: The level 2 processes blocks of eight bytes while level 3 checks one byte at a time. However, when looking at the source code of level 3, we can observe that several obfuscation techniques have been used for creating the native part of the application: opaque predicates, custom comparison functions, and randomness usage. From *OATs'inside* perspective, these obfuscation techniques have no impact on the analysis.

6.3.3 Level 4. The level 4 of the OWASP Uncrackable application is not similar to levels 2 and 3. It embeds several libraries dedicated cryptography or anti-debugging. The password itself is checked after being processed by cryptographic algorithms. The dynamic part of *OATs'inside* has been able to correctly retrieve the Java actions performed by the native code. Nevertheless, the symbolic analysis has not produced any output. Indeed, during the analysis, the symbolic execution has to go through cryptographic functions, which are known to challenge symbolic analysis. In our case, the analysis ends due to lack of internal memory space: The internal stack of the analysis is full. The modification of the internal memory space of angr is left as future work.

6.4 Performance Overhead

To quantify the overhead of the Runner module, we ran an AES-128 over a 16-byte block of data using *OATs'inside* and a Sony Xperia X under AOSP Android 7.0. We used two implementations: one in full Java that stores intermediate results in Java arrays (hereinafter AES-J), and the other is a native implementation manipulating C variables (hereinafter AES-C). AES-J intensively stressed the heap, either from the interpreted version (AES-J DEX) or from the compiled version (AES-J AOTC). Indeed, the ratio between computation and heap accesses was unbalanced in favor of the latter. The results are given in Table 4. The overhead was reasonable for the AES-J DEX implementation and non-existent for the AES-C implementation. For these versions, a lot of the time (68% and 41%) was consumed by protobuf for sending logs to the host. For AES-C, no performance overhead is

[10] *UnCrackable* level 2 source code: https://github.com/OWASP/mstg-crackmes/tree/master/Android/Level2.

Table 4. Time Overhead and Number of Actions/Events for 1000 AES-128 Computations and OWASP UnCrackable App Level 2

			AES-J DEX	AES-J AOTC	AES-C	OWASP lvl 2*
Time (s)	Bare-Metal	Total	0.02	0.1	0.007	0.3
	OATs'inside	Total	121 ×6,050	1,026 ×10,260	0.007 ×1	4.025 ×13
		Protobuf	82 68 %	423 41 %	0 0 %	1.325 33 %
		Runtime	39 32 %	603 59 %	0.007 100 %	2.7 67 %
No. of actions	*OATs'inside*	Allocation	2,000	1,002	0	5,939
		Access	5,287,000	11,385,168	0	4,611
		Methods	6,828,000	6,828,000	0	76,620
Size of actions (Go)	*OATs'inside*	All actions	1.1	1.2	0	0.02
No. of signals	*OATs'inside*	SEGV	0	21,135,992	0	85
		BP	2,000	27,965,993	0	71,371

*: OWASP UnCrackable application level 2 with password filled at startup.
The validation buttons are clicked automatically and then the application quits itself.

observed, because no objects of the Java level are manipulated by the C code, and hence no events are generated. The overhead was much higher for the fully compiled version (AOTC): A factor of 10,260 was observed because of the generation of the SEGV and BP events.

These three cases can be considered as extreme cases: AES-J AOTC generates numerous events and does not use any whitelisted library while AES-C generates nothing. To quantify the overhead on more real cases, we have customized OWASP UnCrackable level 2 so that the application automatically fills a password and exits after checking it. This way, we are able to measure the overhead during the full life of an application. The obtained overhead, execution slowed down by 13 times, is reasonable for an analysis system. This shows the effectiveness of the whitelisting mechanism in real cases.

Figure 8 shows the evolution of the overhead depending on the number of actions. Time is represented with a logarithmic scale. We observed that the overhead was linear with the number of actions. The highest overhead was induced by SEGV actions.

To assess the overhead of the Memory dumper module, we dumped the contents of two applications: a simple "hello world" and the biggest ARMv8-compatible APK from AndroZoo, which was retrieved in 2019 (md5 given in Table 5). The results are shown in Table 5. The time and the size of the dump stay almost the same while the application is $1,000$ times bigger. Indeed, most of the memory contained libraries and areas allocated for all applications.

6.5 *OATs'inside* Robustness against Dedicated Attacks

OATs'inside stealthiness. Obfuscated applications could try avoiding being analyzed. Then, it is important to assess the capacity of *OATs'inside* not being detected. One could argue that the behavior of *OATs'inside* is fingerprintable by detecting the generation of SEGV and TRAP signals. However, they can never be caught by the application, because we capture them. Additionally, *OATs'inside* induces a time overhead when running the application. Then, an application could fingerprint the time of the execution. It could also measure the difference

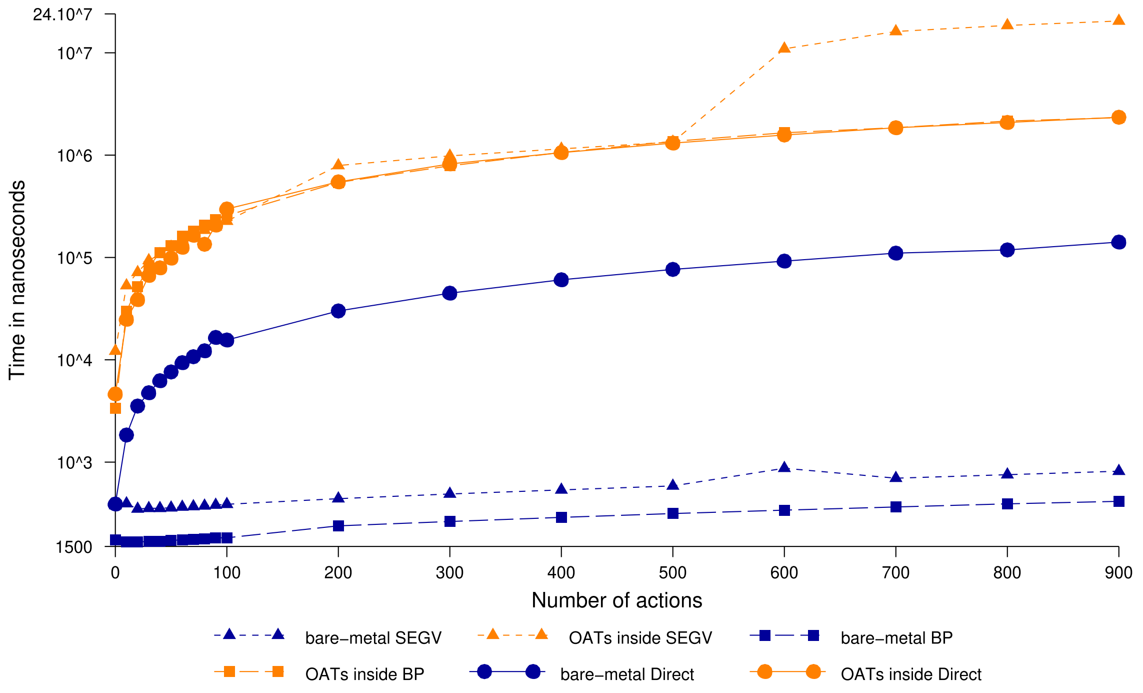


Fig. 8. Runner module overhead with No. of actions.

Table 5. Dump Size Depending on the APK Size

APK name	Hello world	7146b3c02f0f4e3420c4471c2034de9d
APK size	174 Kb	140 Mb
Dump size	1.5 Gb	1.7 Gb
Dump time	8,687	9,257

between the time spent for accessing a variable or a field. Such techniques can be defeated by hooking the syscall `gettimeofday` and changing its return value to a nominal one [36]. Also, a standard way to avoid being debugged is to check that no breakpoints have been set up or that the code has not been modified by using checksums. *OATs'inside* does not modify the application code but rather modifies call and return addresses to redirect them to breakpoints. An application could scan these addresses, trying to detect specifically *OATs'inside*. *OATs'inside* controls the MMU and could redirect the accesses for the breakpoint area to the legitimate code area [34], making them stealth. These questions are left as future work.

Finally, an application could try to check if the running device is a legitimate one, that is, if the phone is rooted or emulated or if analysis systems such as debugging or binary analyzers are installed [23]. Such behaviors have been encountered when analyzing OWASP UnCrackable application level 4. The analysis itself is described in Section 6.3. When looking at its source code, we can note that the application uses a library[11] dedicated to detect rooted devices. *OATs'inside* is not detected by the employed techniques, since its dynamic component is a *legitimate* Android system: It is not different from a custom Android image compiled by a smartphone constructor. Even if these techniques are not targeting *OATs'inside* specifically, bypassing them transparently makes *OATs'inside* more practical to use, since numerous malware use these on-the-shelf analysis countermeasures.

[11] https://github.com/scottyab/rootbeer.

OATs'inside robustness. Additionally to trying to detect *OATs'inside*, malware can try to bypass *OATs'inside* by circumventing the monitoring made by the Runner module. The main idea for the malware would be to modify the functions or the values that are used by *OATs'inside* to perform its dynamic analysis. For instance, the application would hook the logging commands or modify, after *OATs'inside* sets its hooking breakpoints, the methods addresses. The malware can also generates fake events by forging them. In these cases, *OATs'inside* outputs would be incorrect by either missing events or exposing non-existing ones. Similarly to the stealthiness problem, since *OATs'inside* controls the whole system, dedicated solutions could be imagined for each problem. A general solution based on MMU to protect critical points is left as future work.

7 CONCLUSION

This article presents a new methodology combining a dynamic analysis and a concolic execution for capturing all Java behaviors of the native obfuscated part of an application. These behaviors were systematically categorized and, for each behavior, we have designed a dedicated way to retrieve it either online during the execution or offline during the symbolic analysis. We have systematically reviewed state-of-the-art tools against the combination of all possible behaviors with three application formats (DEX, OAT, JNI) and thee types of obfuscation (Packing, native obfuscation using Tigress, DHA). In these cases, *OATs'inside*, achieves the largest coverage of obfuscated behavior retrieval among other related tools. Nevertheless, some advanced obfuscation techniques applied to the native code may fool *OATs'inside* by attacking its symbolic execution engine angr. The output would then contain very complex expressions.

Experiments show that the overhead induced by *OATs'inside* is linear in the number of events that bypass the JNI interface. Combined with implemented optimizations such as Android library white-listing, this result shows the practicality of the proposed approach. We also illustrated the benefit of *OATs'inside* on applications containing anti-debugging techniques provided by the OWASP foundation. We easily recover the hidden password from the native code by identifying the appropriate control flow conditions and the tests that are performed by the native code. With few iterations, we recover the expected password, solving the challenge. It works for two levels of difficulty, the third one being only partially analyzed.

Our proposal is released as an open source tool and patches that modify the Android source code. The benefit of deploying the analysis in the heart of Android is the stealthiness of the analysis: *OATs'inside* is, from the application point of view, another modified Android version, a very common practice in the Android ecosystem. Further work can be conducted in this direction, especially by investigating the countermeasures that an attacker could implement to detect *OATs'inside*. Porting *OATs'inside* to newer versions of Android is also of independent interest for analyzing applications that are only compatible with recent versions of Android.

A OWASP UNCRACKABLE APP LEVEL 3-GENERATED OLCFG

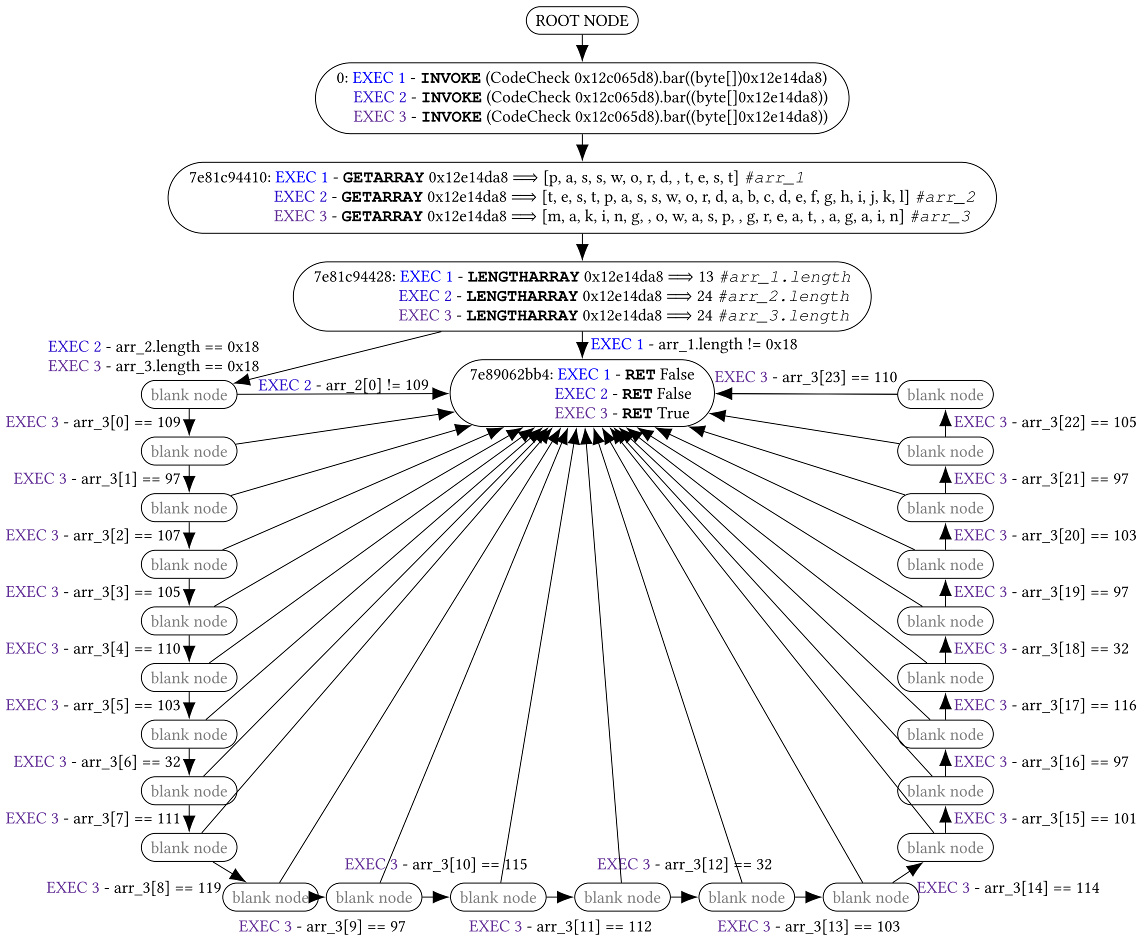

ROOT NODE
0: EXEC 1 - INVOKE (CodeCheck 0x12c065d8).bar((byte[])0x12e14da8)
EXEC 2 - INVOKE (CodeCheck 0x12c065d8).bar((byte[])0x12e14da8))
EXEC 3 - INVOKE (CodeCheck 0x12c065d8).bar((byte[])0x12e14da8))
7e81c94410: EXEC 1 - GETARRAY 0x12e14da8 ⟹ [p, a, s, s, w, o, r, d, , t, e, s, t] #arr_1
EXEC 2 - GETARRAY 0x12e14da8 ⟹ [t, e, s, t, p, a, s, s, w, o, r, d, a, b, c, d, e, f, g, h, i, j, k, l] #arr_2
EXEC 3 - GETARRAY 0x12e14da8 ⟹ [m, a, k, i, n, g, , o, w, a, s, p, , g, r, e, a, t, , a, g, a, i, n] #arr_3
7e81c94428: EXEC 1 - LENGTHARRAY 0x12e14da8 ⟹ 13 #arr_1.length
EXEC 2 - LENGTHARRAY 0x12e14da8 ⟹ 24 #arr_2.length
EXEC 3 - LENGTHARRAY 0x12e14da8 ⟹ 24 #arr_3.length
EXEC 2 - arr_2.length == 0x18
EXEC 3 - arr_3.length == 0x18
EXEC 1 - arr_1.length != 0x18
EXEC 2 - arr_2[0] != 109
7e89062bb4: EXEC 1 - RET False
EXEC 2 - RET False
EXEC 3 - RET True
EXEC 3 - arr_3[23] == 110
EXEC 3 - arr_3[0] == 109
EXEC 3 - arr_3[1] == 97
EXEC 3 - arr_3[2] == 107
EXEC 3 - arr_3[3] == 105
EXEC 3 - arr_3[4] == 110
EXEC 3 - arr_3[5] == 103
EXEC 3 - arr_3[6] == 32
EXEC 3 - arr_3[7] == 111
EXEC 3 - arr_3[8] == 119
EXEC 3 - arr_3[9] == 97
EXEC 3 - arr_3[10] == 115
EXEC 3 - arr_3[11] == 112
EXEC 3 - arr_3[12] == 32
EXEC 3 - arr_3[13] == 103
EXEC 3 - arr_3[14] == 114
EXEC 3 - arr_3[15] == 101
EXEC 3 - arr_3[16] == 97
EXEC 3 - arr_3[17] == 116
EXEC 3 - arr_3[18] == 32
EXEC 3 - arr_3[19] == 97
EXEC 3 - arr_3[20] == 103
EXEC 3 - arr_3[21] == 97
EXEC 3 - arr_3[22] == 105
blank node

Fig. 9. Final olCFG of bar.

Figure 9 shows the final olCFG obtained when analyzing OWASP UnCrackable application level 3 using *OATs'inside*. For the sake of clarity, only three executions are represented. As stated in Section 6.3, this graph is obtained as simply as the one of level 2. The additional obfuscation used by the native code is handled transparently for the analyst. The only difference between the two levels lies in the comparison process: Level 3 compares byte by byte while level 2 checks blocks of 8 bytes.

REFERENCES

[1] Vitor Monte Afonso, Paulo L. de Geus, Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, Giovanni Vigna, Adam Doupé, and Mario Polino. 2016. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *Proceedings of the Network and Distributed System Security Symposium.*

[2] Android. 2018. *Dalvik Executable Format.* Retrieved from https://source.android.com/devices/tech/dalvik/dex-format.

[3] Android. 2019. *Dalvik Bytecode.* Retrieved from https://source.android.com/devices/tech/dalvik/dalvik-bytecode.

[4] ARM. 2017. *ARM Architecture Reference Manual: ARMv8, for ARMv8-A Architecture Profile.*

[5] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. 2017. Artist: The android runtime instrumentation and security toolkit. In *Proceedings of the IEEE European Symposium on Security and Privacy*. IEEE, Los Alamitos, CA, 481–495.

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *Comput. Surv.* 51, 3 (July 2018).

[7] Sebastian Banescu, Christian S. Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC'16)*. 189–200.

[8] Judong Bao, Yongqiang He, and Weiping Wen. 2018. DroidPro: An AOTC-based bytecode-hiding scheme for packing the android applications. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications/IEEE International Conference on Big Data Science and Engineering*. IEEE, Los Alamitos, CA, 624–632.

[9] Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *Comput. Surv.* 17, 4 (December 1985).

[10] Christian Collberg, Sam Martin, Jonathan Myers, Bill Zimmerman, Petr Krajca, Gabriel Kerneis, Saumya Debray, and Babak Yadegari. [n. d.]. The Tigress C Diversifier/Obfuscator. Retrieved from https://tigress.wtf/.

[11] Christian Collberg and Jasvir Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Number 1. Addison-Wesley Professional.

[12] Valerio Costamagna and Cong Zheng. 2016. ARTDroid: A virtual-method hooking framework on android ART runtime. In *Proceedings of the International Workshop on Innovations in Mobile Privacy and Security Co-located with the International Symposium on Engineering Secure Software and Systems*. CEUR Workshop Proceedings, 20–28.

[13] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. 2018. Things you may not know about android (Un) packers: A systematic study based on whole-system emulation. In *Proceedings of the Network and Distributed System Security Symposium.*

[14] 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 *Proceedings of the USENIX Conference on Operating Systems Design and Implementation.* USENIX, 393–407.

[15] Pierre Graux, Jean-Francois Lalande, Pierre Wilke, and Valérie Viet Triem Tong. 2020. Abusing android runtime for application obfuscation. In *Proceedings of the Workshop on Software Attacks and Defenses*. IEEE.

[16] Aric Hagberg, Pieter Swart, and Daniel Chult. 2008. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the Python in Science Conference.* 11–16.

[17] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. 2011. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, New York, NY, 639–652.

[18] Patrik Lantz and Bjorn Johansson. 2015. Towards bridging the gap between Dalvik bytecode and native code during static analysis of Android applications. In *Proceedings of the International Wireless Communications and Mobile Computing Conference*. IEEE, 587–593.

[19] Hongliang Liang, Yudong Wang, Tianqi Yang, and Yue Yu. 2018. AppLance: A lightweight approach to detect privacy leak for packed applications. In *Proceedings of the Nordic Conference on Secure IT Systems*. Springer, 54–70.

[20] Yibin Liao, Jiakuan Li, Bo Li, Guodong Zhu, Yue Yin, and Ruoyan Cai. 2016. Automated detection and classification for packed android applications. In *Proceedings of the International Conference on Mobile Services*. IEEE, 200–203.

[21] Carey Nachenberg. 1996. Understanding and managing polymorphic viruses. *The Symantec Enterprise Papers* 30 (1996).

[22] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 89–100.

[23] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the ACM European Workshop on System Security*. ACM, 5.

[24] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin T. S. Chan. 2014. On tracking information flows through JNI in Android applications. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 180–191.

[25] Ali Razeen, Alvin R. Lebeck, David H. Liu, Alexander Meijer, Valentin Pistol, and Landon P. Cox. 2018. SandTrap: Tracking information flows on demand with parallel permissions. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 230–242.

[26] Rolf Rolles. 2009. Unpacking virtualization obfuscators. In *Proceedings of the USENIX Workshop on Offensive Technologies*. USENIX, Montreal, Canada.

[27] Paul Sabanal. 2015. Hiding behind ART. In *Proceedings of the Black Hat Asia*.

[28] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE.

[29] Mahendra Pratap Singh and Manoj Kumar Jain. 2014. Evolution of processor architecture in mobile phones. *Int. J. Comput. Appl.* 90, 4 (March 2014).

[30] Nikolaos Totosis and Constantinos Patsakis. 2018. Android hooking revisited. In *IEEE International Conference on Dependable, Autonomic and Secure Computing, International Conference on Pervasive Intelligence and Computing, International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress*. IEEE, 552–559.

[31] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. 2015. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 659–673.

[32] 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 *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1137–1150.

[33] Michelle Y. Wong and David Lie. 2016. IntelliDroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the Network and Distributed System Security Symposium*. 21–24.

[34] Michelle Y. Wong and David Lie. 2018. Tackling runtime-based obfuscation in Android with TIRO. In *Proceedings of the USENIX Security Symposium*. USENIX, 1247–1262.

[35] Rafael J. Wysocki. 2007. Freezing of Tasks. Retrieved from https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/power/freezing-of-tasks.txt?h=v3.10.

[36] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive unpacking of Android apps. In *Proceedings of the International Conference on Software Engineering*. IEEE, 358–369.

[37] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards on-device non-invasive mobile malware analysis for ART. In *Proceedings of the USENIX Security Symposium*. USENIX, 289–306.

[38] Lok-Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the USENIX Security Symposium*. USENIX, 569–584.

[39] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. 2015. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*. Springer, 359–381.

[40] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. 2015. Dexhunter: Toward extracting hidden code from packed android applications. In *Proceedings of the European Symposium on Research in Computer Security*. Springer, 293–311.

Received 23 April 2022; revised 19 September 2022; accepted 26 January 2023