usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART

Lei Xue, *The Hong Kong Polytechnic University;* Yajin Zhou, *unaffiliated;*
Ting Chen, *University of Electronic Science and Technology of China;*
Xiapu Luo, *The Hong Kong Polytechnic University;* Guofei Gu, *Texas A&M University*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xue

This paper is included in the Proceedings of the
26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART

Lei Xue[†], Yajin Zhou[♮], Ting Chen[†‡], Xiapu Luo[†*], Guofei Gu[§]
[†]*Department of Computing, The Hong Kong Polytechnic University*
[♮]*Unaffiliated*
[‡]*Cybersecurity Research Center, University of Electronic Science and Technology of China*
[§]*Department of Computer Science & Engineering, Texas A&M University*

Abstract

It's an essential step to understand malware's behaviors for developing effective solutions. Though a number of systems have been proposed to analyze Android malware, they have been limited by incomplete view of inspection on a single layer. What's worse, various new techniques (e.g., packing, anti-emulator, etc.) employed by the latest malware samples further make these systems ineffective. In this paper, we propose `Malton`, a novel on-device non-invasive analysis platform for the new Android runtime (i.e., the ART runtime). As a dynamic analysis tool, `Malton` runs on real mobile devices and provides a comprehensive view of malware's behaviors by conducting multi-layer monitoring and information flow tracking, as well as efficient path exploration. We have carefully evaluated `Malton` using real-world malware samples. The experimental results showed that `Malton` is more effective than existing tools, with the capability to analyze sophisticated malware samples and provide a comprehensive view of malicious behaviors of these samples.

1 Introduction

To propose effective solutions, it is essential for malware analysts to fully understand malicious behaviors of Android malware. Though many systems have been proposed, malware authors have quickly adopted advanced techniques to evade the analysis. For instance, since the majority of static analysis tools inspect the Dalvik bytecode [2], malware circumvent them by using various obfuscation techniques to raise the bar of code comprehension [61], implementing malicious activities in native libraries to evade the inspection [13, 59, 70, 92], and leveraging packing techniques to hide malicious payloads [82, 85, 88]. For example, the percentage of packed Android malware has increased from 10% to 25% [36], and 37.0% of the Android apps execute native code [11].

These sophisticated techniques employed by the latest malware also make the dynamic analysis systems ineffective. First, malicious behaviors usually cross several system layers (e.g., the Android runtime, the Android framework, and native libraries, etc.). However, the majority of dynamic analysis systems [34, 46, 73, 91] lack of the capability of cross-layer inspection, and thus provide incomplete view of malicious behaviors. For example, CopperDroid [73] monitors malware behaviors mainly through the trace of system calls (e.g., *sys_sendto()* and *sys_write()*). Thus, it is hard to expose the execution details in the Android framework layer and the runtime layer, due to the well-known semantic gap challenge. Second, the anti-debug and anti-emulator techniques employed by malware [44, 47, 56, 74] as well as the new Android runtime (i.e., the ART runtime) further limit the usage of many dynamic analysis systems. For example, in [14], 98.6% malware samples were successfully analyzed on the real smartphone, whereas only 76.84% malware samples were successfully inspected using the emulator. Most of the existing tools either rely on emulators (e.g., DroidScope [83]) or modify the old Android runtime (i.e., Dalvik Virtual Machine, or DVM for short) to monitor malware behaviors (e.g., TaintDroid [38]). Third, it is a common practice that malware executes different payloads according to the commands from the remote command and control (i.e., C&C) servers. However, existing systems are not effective in capturing the execution of all malicious payloads, because they are impaired by the inherent limitation of dynamic analysis (i.e., low code coverage) and the lack of efficient code path exploration technique.

In this paper, we propose `Malton`, a novel on-device non-invasive analysis platform for the ART runtime. Compared with other systems, `Malton` employs two important capabilities, namely, a) multi-layer monitoring and information flow tracking, and b) efficient path ex-

[*] The corresponding author.

ploration, to provide a comprehensive view of malware behaviors. Moreover, Malton does not need to modify malware's bytecode for conducting static instrumentation. To our best knowledge, Malton is the *first* system with such capabilities. Table 7 in Section 6 illustrates the key differences between Malton and other systems.

Malton inspects Android malware on different layers. It records the invocations of Java methods, including sensitive framework APIs and the concerned methods of the malware, in the *framework layer*, and captures stealthy behaviors, such as dynamic code loading and JNI reflection, in the *runtime layer*. Moreover, it monitors library APIs and system calls in the *system layer*, and propagates taint tags and explores different code paths in the *instruction layer*. However, multi-layer monitoring is not enough to provide a comprehensive view of malware behaviors, because malicious payloads could be conditionally executed. We deal with this challenge with the capability to efficiently explore code paths. First, to trigger as many malicious payloads as possible, we propose a multi-path exploration engine based on the concolic execution [27] to generate concrete inputs for exploring different code paths. Second, to conduct efficient path exploration on mobile devices with limited computational resources, we propose an offloading mechanism to move heavy-weight tasks (e.g., solving constraints) to resourceful desktop computers, and an in-memory optimization mechanism that makes the execution flow return to the entry point of the interested code region immediately after exiting the code region. Third, in case the constraint solver fails to find a solution to explore a code path, we equip Malton with a direction execution engine to forcibly execute a specified code path. Since Malton requires the necessary human annotations of the interested code regions, it is most useful in the human-guided detailed exploration of Android malware.

We have implemented a prototype of Malton based on the binary instrumentation framework Valgrind [53]. Since both the app's code and the framework APIs are compiled into native code in the ART runtime, we leverage the instrumentation mechanism of Valgrind to introspect apps and the Android framework. We evaluated Malton with real-world malware samples. The experimental results show that Malton can analyze sophisticated malware samples and provide a comprehensive view of their malicious behaviors.

In summary, we make the following contributions.

- We propose a novel Android malware analysis system with the capability to provide a comprehensive view of malicious behaviors. It has two major capabilities, including multi-layer monitoring and information flow tracking, and efficient path exploration.
- We implement the system named Malton by solving several technical challenges (e.g., cross-layer taint propagation, on-device Java method tracking, execution path exploration, etc.). To the best of our knowledge, it is the *first* system having such capabilities. To engage the whole community, we plan to release Malton to the community.
- We carefully evaluate Malton with real-world malware samples. The results demonstrated the effectiveness of Malton in analyzing sophisticated malware.

The rest of this paper is organized as follows. Section 2 introduces background knowledge and describes a motivating example. Section 3 details the system design and implementation. Section 4 reports the evaluation results. Then, we discuss Malton's limitations and possible solutions in Section 5. After presenting the related work in Section 6, we conclude the paper in Section 7.

2 Background

2.1 The ART Runtime

ART is the new runtime introduced in Android version 4.4, and becomes the default runtime from version 5.0. When an app is being installed, its Dalvik bytecode in the Dex file is compiled to native code[1] by the dex2oat tool, and a new file in the OAT format is generated including both the Dalvik bytecode and native code. The OAT format is a special ELF format with some extensions.

The OAT file has an oatdata section, which contains the information of each class that has been compiled into native code. The native code resides in a special section with the offset indicated by the oatexec symbol. Hence, we can find the information of a Java class in the oatdata section and its compiled native code through the oatexec symbol.

When an app is launched, the ART runtime parses the OAT file and loads the file into memory. For each Java class object, the ART runtime has a corresponding instance of the C++ class Object to represent it. The first member of this instance points to an instance of the C++ class Class, which contains the detailed information of the Java class, including the fields, methods, etc. Each Java method is represented by an instance of the C++ class ArtMethod, which contains the method's address, access permissions, the class to which this method belongs, etc. The C++ class ArtField is used to represent a class field, including the class to which this field belongs, the index of this field in its class, access rights, etc. We can leverage the C++ Object, Class, ArtMethod and ArtField to find the detailed information of the Java class, methods and fields of the Java class.

[1] Native code denotes the native instructions that could directly run with a particular processor.

Listing 1: A motivating example

```
1  public static native void readContact();
2  public static native void parseMSG(String msg);
3  private void readIMSI(){
4    TelephonyManager telephonyManager =
5      (TelephonyManager) getSystemService(
6        Context.TELEPHONY_SERVICE);
7    String imsi = telephonyManager.getSubscriberId();
8    // Send back data through SMTP protocol
9    smtpReply(imis);
10 }
11 private void procCMD(int cmd, String msg){
12   if(cmd == 1) {
13     readSMS();  // Read SMS content
14   } else if(cmd == 2) {
15     readContact();  // Read Contact content
16   } else if(cmd == 3) {
17     readIMSI();  // Read device IMSI information
18   } else if(cmd == 4) {
19     rebootDevice();  // Reboot the device
20   } else if(cmd == 5) {
21     parseMSG(msg);  // Parse msg in native code
22   } else { // The command is unrconginized.
23     reply("Unknown command!");
24   }
25 }
26 public boolean equals(String s1, String s2) {
27   if(s1.count != s2.count)
28     return false;
29   if(s1.hashCode() != s2.hashCode())
30     return false;
31   for(int i = 0; i < count; ++i)
32     if (s1.charAt(i) != s2.charAt(i))
33       return false;
34   return true;
35 }
36 public void onReceiver(Context context, Intent intent){
37   String body = smsMessage.getMessageBody();
38   // Get the telephone of the sender
39   String sender = smsMessage.getOriginatingAddress();
40   // Check if the SMS is sent form the controller
41   if(equals(sender, "6223**60")) {
42     procCMD(Interger.parseInt(body), body);
43   }
44   ...
45 }
```

The Android framework is compiled into an OAT file named "*system@framework@boot.oat*". This file is loaded to the fixed memory range for all apps running on the device without ASLR enabled [69].

2.2 Motivating Example

We use the example in Listing 1 to illustrate the usage of `Malton`. In this example, the method *onReceiver()* is an SMS listener and it is invoked when an SMS arrives. In this method, the telephone number of the sender is first acquired (Line 39) for checking whether the SMS is sent from the controller (Tel: 6223**60). Only the SMS from the controller will be processed by the method *procCMD()* (Line 42). There are 5 types of commands, each of which leads to a special malicious behavior (i.e., Line 13, 15, 17, 19 and 21). Reading contact and parsing SMS are implemented in the JNI methods *readContact()* (Line 1) and *parseMSG()* (Line 2), respectively.

Existing malware analysis tools could not construct a complete view of the malicious behaviors. For example, when *cmd* equals 3 (Line 16), IMSI is obtained by invoking the framework API *getSubscriberId()* (Line 7), and then leaked through SMTP protocol (Line 9). Although existing tools (e.g., CopperDroid [73]) can find that the malware reads IMSI and leaks the information by system call *sys_sendto()*, they cannot locate the method used to get IMSI and how the IMSI is leaked in detail, because *sys_sendto()* can be called by many functions (e.g., JavaMail APIs, Java Socket methods and C/C++ Socket methods) from both the framework layer and the native layer. `Malton` can solve this problem because it performs multi-layer monitoring.

When *cmd* equals 5, the content of SMS, which is obtained from the framework layer (Line 37), will be parsed in the JNI method *parseMSG()* (Line 2) by native code. Although taint analysis could identify this information flow, existing static instrumentation based tools (e.g., TaintART [71] and ARTist [21]) cannot track the information flow in the native code. `Malton` can tackle this issue since it offers cross-layer taint analysis.

Moreover, as shown in the method *procCMD()* (Line 11), the malware performs different activities according to the parameter *cmd*. Due to the low code coverage of dynamic analysis, how to efficiently explore all the malicious behaviors with the corresponding inputs is challenging. `Malton` approaches this challenge by conducting concolic execution with in-memory optimization and direct execution. Furthermore, we propose a new offloading mechanism to avoid overloading the mobile devices with limited computational resources. Since some constraints may not be solved (e.g., the hash functions at Line 29), we develop a direct execution engine to cover specified branches forcibly.

3 Design and Implementation

In this section, we first illustrate the design of our approach, and then detail the implementation of `Malton`.

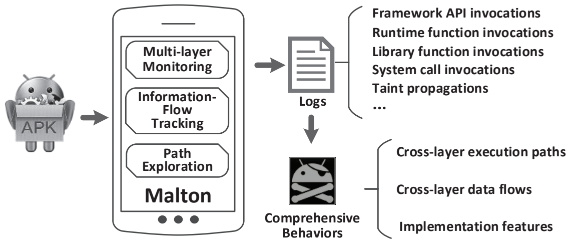


Figure 1: The scenario of `Malton`.

3.1 Overview

`Malton` helps security analysts obtain a complete view of malware samples under examination. To achieve this goal, `Malton` supports three major functionalities. First, due to the multi-layer nature of Android system, `Malton` can capture malware behaviors at different layers. For instance, malware may conduct malicious activities by invoking native code from Java methods, and such behaviors involve method invocations and data transmission at multiple layers. The challenging issue is how to effectively bridge the semantic gap when monitoring the ARM instructions.

Second, malware could leak private information by passing the data across multiple layers back and forth. Note that many framework APIs are JNI methods (e.g., *String.concat()*, *String.toCharArray()*, etc.), whose real implementations are in native code. `Malton` can detect such privacy leakage because it supports cross-layer information flow tracking (Section 3.5).

Third, since malware may conduct diverse malicious activities according to different commands and contexts, `Malton` can trigger these activities by exploring the paths automatically (Section 3.6). It is non-trivial to achieve this goal because dynamic analysis systems usually have limited code coverage.

Figure 1 illustrates a use scenario of `Malton`. `Malton` runs in real Android devices and conducts multi-layer monitoring, information flow tracking, and path exploring. After running a malware sample, `Malton` generates logs containing the information of method invocations and taint propagations at different layers and the result of concolic executions. Based on the logs, we can reconstruct the execution paths and the information flows for characterizing malware behaviors.

Though `Malton` performs the analysis in multiple layers as shown in Figure 2, the implementation of `Malton` in each layer is not independent. Instead, different layers share the information with each other. For example, the taint propagation module in the instruction layer needs the information about the Java methods that are parsed and processed in the framework layer.

`Malton` is built upon Valgrind [53] (V3.11.0) with around 25k lines of C/C++ codes calculated by CLOC [1]. Next, we will detail the implementation at each layer.

3.2 Android Framework Layer

To monitor the invocations of privacy-concerned Java methods of the Android framework and the app itself, `Malton` instruments the native code of the framework and the app. Since the Dalvik code has been compiled into native instructions, we leverage Valgrind for the instrumentation. The challenge here is how to recover and understand the semantic information of Java methods from the ARM instructions, including the method name, parameters, call stacks, etc. For instance, if a malware sample uses the Android framework API to retrieve user contacts, `Malton` should capture this behavior from the ARM instructions and recover the context of the API invocation. To address this challenge, we propose an efficient way to bridge the semantic gaps between the low level native instructions and upper layer Java methods.

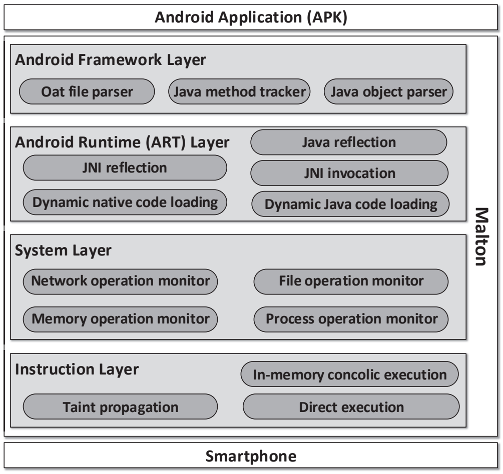


Figure 2: The overview of `Malton`.

Java Method Tracker To track the Java method invocations, we need to identify the entry point and exit points of each Java method from the ARM instructions dynamically. Note that the ARM instructions resulted from the Dalvik bytecode are further translated into multiple IR blocks by `Malton`. An IR block is a collection of IR statements with one entry point and multiple exit points. One exit point of an IR block could be either the conditional exit statement (i.e., `Ist_Exit`) or the next statement (i.e., `Ist_Next`). We leverage the APIs from Valgrind to add instrumentation at the beginning, before any IR instruction, after any IR instruction, or at the end of the selected IR block. The instrumentation statements will invoke our helper functions.

To obtain the entry point of a Java method, we use the method information in the OAT file. Specifically, the OAT file contains the information of each compiled Java method (`ArtMethod`), including the method name, offset of the ARM instructions, access flags, etc. `Malton` parses the OAT files of both the Android framework and the app itself to retrieve such information, and keeps it in a hash table. When the native code is translated into the IR blocks, `Malton` looks up the beginning address of each IR block in the hash table to decide whether it is the entry of a Java method. If so, `Malton` inserts the helper function (i.e., *callTrack()*) at the beginning of the block to record the method invocation and parse arguments when it is executed.

To identify the exit point of a Java method, `Malton` leverages the method calling convention of the ARM architecture[2]. Specifically, the return address of a method is stored in the link register (i.e., the lr register) when the method is invoked. Hence, in *callTrack()*, `Malton` pushes

[2] Comments in file /art/compiler/dex/quick/arm/arm_lir.h

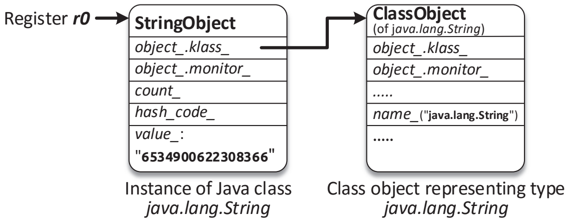


Figure 3: The example of parsing the Java object of the result of *TelephonyManager.getDeviceId()*

the value of *lr* into the method call stack since *lr* could be changed during the execution of the method. `Malton` also inserts the helper function (i.e., *retTrack()*) before each exit point (i.e., `Ist_Exit` and `Ist_Next`) of the IR block. In *retTrack()*, `Malton` compares the jump target of the IR block with the method's return address stored at the top of the method call stack. If they are equal, an exit point of the method is found, and this return address is popped from the method call stack.

`Malton` parses the arguments and the return value of the method after the entry point and the exits point of the method are identified, respectively. According to the method calling convention, the register *r0* points to the `ArtMethod` object of current method, and registers $r1 - r3$ contain the first three arguments. Other arguments beyond the first three words are pushed into the stack by the caller. For example, when the framework method *sendMessageAtTime(Message msg, long uptimeMillis)* of class *android/os/Handler* is invoked, *r0* points to the `ArtMethod` instance of the method *sendMessageAtTime()*, *r1* stores the *this* object and *r2* represents the argument *msg*. For the argument *uptimeMillis*, the high 32 bits are stored in the register *r3* and the low 32 bits are pushed into the stack. When the method returns, the return value is stored in the register *r0* if the return value is 32 bits, and in registers *r0* and *r1* if the return value is 64 bits.

Java Object Parser After getting the method arguments and the return value, we need to further parse the value if it is not the primitive data. There are two major data types [42] in Java, including primitive data types and reference/object data types (objects). For the primitive types, which include byte, char, short, int, long, float, double and boolean, we can directly get the value from registers and the stack. For the object, the value that we obtain from the register or the stack is a pointer that points to a data structure containing the detailed information of this object. Following this pointer, we get the class information of this object, and then parse the memory of this object to determine the concrete value.

Figure 3 illustrates the process of parsing the Java object of the result of *TelephonyManager.getDeviceId()*. According to its method shorty, we know that the return value of this API is a Java object represented by an *Object* instance, of which the memory address is stored in the register *r0*. Then, we can decide that the concrete type of this object is *java.lang.String*. By parsing the results according to the memory layout of *String* object, which is represented by the *StringObject* data structure, we can obtain the concrete string "6534900622308366". Currently, `Malton` can parse the Java objects related to String and Array. To handle new objects, users just need to implement the corresponding parsers for `Malton`.

3.3 Android Runtime Layer

To capture stealthy behaviors that cannot be monitored by the Java method tracker in the Android framework layer, `Malton` further instruments the ART runtime (i.e., *libart.so*). For example, the packed malware may use the internal functions of the ART runtime to load and execute the decrypted bytecode directly from the memory [85,88]. Malicious payloads could also be implemented in native code, and then invoke the privacy-concerned Java methods from native code through the JNI reflection mechanism. While the invoked Java method could be tracked by the Java method tracker in the Android framework layer, `Malton` tracks the JNI reflection to provide a comprehensive view of malicious behaviors, such as, the context when privacy-concerned Java methods are invoked from the native code. This is one advantage of `Malton` over other tools.

Table 1: Runtime behaviors related functions.

Behavior	Functions
Native code loading	*JavaVMExt::LoadNativeLibrary()*

Java code loading	*DexFile::DexFile()*
	DexFile::OpenMemory()
	ClassLinker::DefineClass()

JNI invocation	*artFindNativeMethod()*
	ArtMehod::invoke()

JNI reflection	*InvokeWithVarArgs()*
	InvokeWithJValues()
	InvokeVirtualOrInterfaceWithJValues()
	InvokeVirtualOrInterfaceWithVarArgs()

Java reflection	*InvokeMethod()*

Table 1 enumerates the runtime behaviors and the corresponding functions in the ART runtime that `Malton` instruments. Native code loading means that malicious code could be implemented in native code and loaded into memory, where Java code loading refers to loading the Dalvik bytecode. Note that Android packers usually exploit these APIs to directly load the decrypted bytecode from memory. JNI invocation refers to all the function calls from Java methods to native methods. This includes the JNI calls in the app and the Android framework. JNI reflection, on the other hand, refers to calling Java methods from native code. For instance, malicious payloads implemented in native code could invoke framework APIs using JNI reflection. Java reflection is commonly used by malware to modify the runtime behavior for evading the static analysis [61]. For example, framework APIs could be invoked by decrypting the method names and class names at runtime using Java reflection.

3.4 System Layer

Malton tracks system calls and system library functions at the system layer. To track system calls, Malton registers callback handlers before and after the system call invocation through Valgrind APIs. For system library functions, Malton wraps them using the function wrapper mechanism of Valgrind. In the current prototype, Malton focuses on four types of behaviors at the system lever.

- Network operations. Since malware usually receives the control commands and sends private data through network, Malton inspects these behaviors by wrapping network related system calls, such as, *sys_connect()*, *sys_sendto()*, *recvfrom()*, etc.
- File operations. As malware often accesses sensitive information in files and/or dynamically loads malicious payloads from the file system, Malton records file operations to identify such behaviors.
- Memory operations. Since packed malware usually dynamically modifies its own codes through memory operations, like *sys_mmap()*, *sys_protect()*, etc., Malton monitors such memory operations.
- Process operations. As malware often needs to fork new process, or exits when the emulator or the debug environment is detected, Malton captures such behaviors by monitoring system calls relevant to the process operations, including *sys_execve()*, *sys_exit()*, etc.

Moreover, Malton may need to modify the arguments and/or the return values of system calls to explore code paths. For example, the C&C server may have been shut down when malware samples are being analyzed. In this case, Malton replaces the results of the system call *sys_connect()* to success, or replaces the address of C&C with a bogus one controlled by the analyst to trigger malicious payloads. We will discuss the techniques used to explore code paths in Section 3.6.

3.5 Instruction Layer: *Taint Propagation*

At the instruction layer, Malton performs two major tasks, namely, taint propagation and path exploration. Note that accomplishing these tasks needs the semantic information in the upper layers, such as the method invocations for identifying the information flow, etc.

Table 2: The taint propagation related IR Statements.

IR Statement	Representation
Ist_WrTmp	Assign a value (i.e., IR Expression) to a temporary.
Ist_LoadG	Load a value to a temporary with guard.
Ist_CAS	Do an atomic compare-and-swap operation.
Ist_LLSC	Do an either load-linked or store-conditional operation.
Ist_Put	Write a value to a guest register.
Ist_PutI	Write a value to a guest register at a non-fixed offset in the guest state.
Ist_Store	Write a value to memory.
Ist_StoreG	Write a value to memory with guard.
Ist_Dirty	Call a C function.

To propagate taint tags across different layers, Malton works at the instruction layer because the codes of all upper layers become ARM instructions during execution. Since these ARM instructions will be translated into IR statements [53], Malton performs taint propagation on IR statements with byte precision by inserting helper functions before selected IR statements.

Table 3: Taint propagation related IR expressions.

IR Expression	Representation
Iex_Const	A constant-valued expression.
Iex_RdTmp	The value held by a VEX temporary.
Iex_ITE	A ternary if-then-else operation.
Iex_Get	Get the value held by a guest register at a fixed offset.
Iex_GetI	Get the value held by a guest register at a non-fixed offset.
Iex_Unop	A unary operation.
Iex_Binop	A binary operation.
Iex_Triop	A ternary operation.
Iex_Qop	A quaternary operation.
Iex_Load	Load the value stored in memory.
Iex_CCall	A call to a pure (no side-effects) helper C function.

For Malton, there are 9 types IR statements related to the taint propagation, which are listed in Table 2. For the Ist_WrTmp statement, since the source value may be the result of an IR expression, we also need to parse the logic of the IR expression for taint propagation. The IR expressions that can affect the taint propagation are summarized in Table 3. During the execution of the target app, Malton parses the IR statements and expressions in the helper functions, and propagates the taint tags according to the logic of the IR statements and expressions.

Malton supports taint sources/sinks in different layers (i.e., the framework layer and the system layer). For example, Malton can take the arguments and results of both Java methods and C/C++ methods as the taint sources, and check the taint tags of the arguments and the results of sink methods. By default, at the framework layer, 11 types of information are specified as taint sources, including device information (i.e., IMSI, IMEI, SN and phone number), location information (i.e., GPS location, network location and last seen location) and personal information (i.e., SMS, MMS, contacts and call logs). Malton also checks the taint tags of the arguments and results when each framework method is invoked. In the system layer, Malton takes system calls *sys_write()* and *sys_sendto()* as taint sinks by default, because the sensitive information is usually stored to files or leaked out of the device through these system calls. As malware can receive commands from network, `Malton` takes system call *sys_recvfrom()* as the taint source by default. Note that `Malton` can be easily extended to support other methods as taint sources and sinks in both the framework layer and the system layer.

3.6 Instruction Layer: *Path Exploration*

Advanced malware samples usually execute malicious payloads according to the commands received from the C&C server or the special context (e.g., date, locations, etc.). To trigger as many malicious behaviors as possible for analysis, `Malton` employs the efficient path exploration technique, which consists of taint analysis, in-memory concolic execution with an offloading mechanism, and direct execution engine. Specifically, taint analysis helps the analyst identify the code paths depending on the inputs, such as network commands, and the concolic execution module can generate the required inputs to explore the interested code paths. When the inputs cannot be generated, we rely on the direct execution engine to forcibly execute certain code paths. Since concolic execution [27] is a well-known technique in the community, we will not introduce it in the following. Instead, we detail the offloading mechanism and the in-memory optimization used in the concolic execution module, and explain how the direct execution engine works.

Concolic Execution: Offloading Mechanism It is nontrivial to apply concolic execution in analyzing Android malware on real devices, because concolic execution requires considerable computational resources, resulting in unacceptable overhead on the mobile devices. To alleviate this limitation, `Malton` utilizes an offloading mechanism that moves the task of solving constraints to the resourceful desktop computers, and then sends back the satisfying results to the mobile devices as inputs. Our approach is motivated by the fact that the time consumption for solving constraints occupies the overall runtime of concolic execution. For example, the percentage of time used to solve constraints is nearly 41% of the KLEE system, even after optimizations [25].

More precisely, when the malware sample is running in our system, `Malton` redirects all the constraints to the logcat messages [4], which could be retrieved by the desktop computer using the ADB (Android Debug Bridge) tool. Then, the constraint solver, which is implemented based on Z3 [33], generates the satisfying inputs and feeds the inputs back to `Malton` through a file. Since we may have multiple code paths that need to be explored, this process could be repeated several times until the constraint solver pushes an empty input file to the device for notifying `Malton` to finish path exploring.

Concolic Execution: In-memory Optimization To speed up the analysis, especially when there are multiple execution paths, each of which depends on the special input, we propose in-memory optimization to restrict concolic execution within the interested code region specified by the analyst without repeatably running from the beginning of the program. By default, the analyst is required to specify the arguments or variables as the input of the concolic execution, which will be represented as symbolic values during concolic execution. For example, the analyst can select the SMS content acquired from the method *getMessageBody()* (Line 37 in Listing 1) as the input. Moreover, the analyst can select the IR statement that lets the input have concrete values as the entry point of the code region, and choose the exit statement (i.e., `Ist_Exit`) or the next statement (i.e., `Ist_Next`) of the subroutine as the exit point of the code region.

`Malton` runs the malware sample until the exit point of the interested code region for collecting constraints and generating new inputs for different code paths through an SMT solver. Then, it forces the execution to return to the entry point of the code region through modifying the program counter and feeds the inputs by writing the new inputs directly into the corresponding locations (i.e., memories or registers). Moreover, `Malton` needs to recover the execution context and the memory state at the entry point of the code region.

To recover the execution context, `Malton` conducts instrumentation at the beginning of the code region, and inserts a helper function to save the execution context (i.e., register states at the first iteration). After that, the saved register states will be recovered in the later iterations. As Valgrind uses the structure `VexGuestArchState` to represent the register states, we save and recover the register states by reading and writing the `VexGuestArchState` data in the memory.

To recover the memory states, `Malton` replaces the system's memory allocation/free functions with our customized implementations to monitor all the memory allocation/free operations. `Malton` can also free the allocated memory or re-allocate the freed memory. Besides, `Malton` inserts a helper function before each memory store (i.e., `Ist_Store` and `Ist_StoreG`) statement to track the memory modifications, so that all the modified memory could be restored.

Alternatively, the analyst can choose the target code region according to the method call graph, or first use static analysis tool to identify code paths and then select a portion of the path as the interested code region.

Direct Execution The concolic execution may not be able to explore all the code paths of the interested code region, because the constraint solver may not find satisfying inputs for complex constraints, such as float-point operations and encryption routines. For the conditional branches with unresolved constraints, Malton has the capability to directly execute certain code paths.

The direct execution engine of Malton is implemented through two techniques: a) modifying the arguments and the results of methods, including library functions, system calls and Java methods; b) setting the guard value of the conditional exit statement (i.e., Ist_Exit). The guard value is the expression used in the Ist_Exit statement to determine whether the branch should be taken.

It's straightforward to modify arguments and the return values of library functions and system calls by leveraging Valgrind APIs. However, it's challenging to deal with the Java methods because there is no interface in Valgrind to wrap Java methods. Fortunately, we have obtained the entry point and exit points of the compiled Java method in the framework layer (Section 3.2). Hence, we could wrap the Java method by adding instrumentation at its entry point and exit points. For example, to change the source telephone number of a received SMS to explore certain code path (Line 41 in Listing 1), Malton can wrap the framework API *SmsMessage.getOriginatingAddress()* and modify its return value to a desired number at the exit points.

To set the guard value of the Ist_Exit statement, we insert a helper function before each Ist_Exit statement and specify the guard value to the result of the helper function. In an IR block, the program can only conventionally jump out of the IR block at the location of the Ist_Exit statement (e.g., an if-branch in the program). The Ist_Exit statement is defined with the format "if(*t*) goto <*dst*>" in Valgrind, where *t* and *dst* represent the guard value and destination address, respectively. By returning "1" or "0" in the helper function, we can let *t* satisfy or dissatisfy the condition for exploring different code paths.

Table 4: Comparison of the capability of capturing the sensitive behaviors of malware samples.

Behavior	CopperDroid	DroidBox	Malton
Personal Info	435 (85.0%)	135 (26.4%)	511 (99.8%)
Network access	351 (68.5%)	211 (41.2%)	445 (86.9%)
File access	438 (85.5%)	509 (99.4%)	512 (100%)
Phone call	52 (10.1%)	1 (0.2%)	59 (11.5%)
Send SMS	26 (5.1%)	15 (2.9%)	28 (5.5%)
Java code loading	NA	509 (99.4%)	512 (100%)
Anti-debugging	4 (0.8%)	NA	4 (0.8%)
Native code loading	NA	NA	160 (31.2%)

4 Evaluation

We evaluate Malton using real-world Android malware samples to answer the following questions.

Q1: Can Malton capture more sensitive operations than other systems?

Q2: Can Malton analyze sophisticated malware samples (e.g., packed malware) to provide a comprehensive view of malicious behaviors?

Q3: Is the path exploration mechanism effective and efficient?

4.1 Sensitive Behavior Monitoring

To answer **Q1**, we compare Malton's capability of capturing sensitive behaviors with CopperDroid [73] and DroidBox [34]. These two systems are implemented by instrumenting Android emulator and modifying the Android system, respectively. Since CopperDroid's website[3] has just queued all our uploaded malware samples, we cannot obtain the corresponding analysis results. Therefore, we downloaded the analysis reports of $1,362$ malware samples that have been analyzed by CopperDroid. According to their md5s, we collected 512 samples, and run them using Malton and DroidBox, respectively. The comparison results are listed in Table 4. The first column shows the type of sensitive behaviors, and the following columns list the numbers and percentages of malware samples that have been detected by each system due to the corresponding sensitive behaviors . We can see that for all the sensitive behaviors Malton detected more samples than the other two systems.

We further manually analyze the malware samples to understand why Malton detects more sensitive behaviors in those samples than the other two systems. First, Malton monitors malware's behaviors in multiple layers, and thus it can capture more behaviors than the systems focusing on one layer. For instance, the malware sample[4] retrieves the serial number and operator information of the SIM card through the framework APIs *TelephonyManager.getSimSerialNumber()* and *TelephonyManager.getSimOperator()*, respectively. However, CopperDroid does not support reconstructing such behaviors from system calls and DroidBox does not monitor these framework APIs. Second, Malton runs on real devices, and hence it could circumvent many anti-emulator techniques. For instance, the malware sample[5] detects the existence of emulator based on the value of *android_id* and *Build.DEVICE*. If the obtained value indicates that it is running in an emulator, the malicious behaviors will not be triggered.

Note that these samples were analyzed by CopperDroid before 2015 and it is likely that their C&C servers were active at that time. However, not all C&C servers

[3] http://copperdroid.isg.rhul.ac.uk/copperdroid/reports.php

[4] md5: 021cf5824c4a25ca7030c6e75eb6f9c8

[5] md5: a0000a85a2e8e458660c094ebedc0c6e

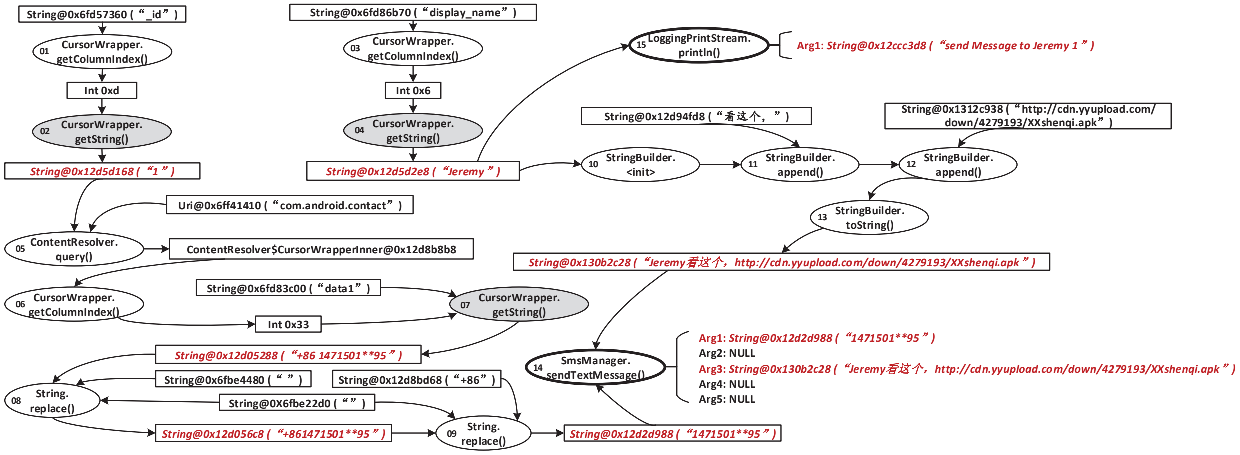

String@0x6fd57360 ("_id")
01 CursorWrapper. getColumnIndex()
Int 0xd
02 CursorWrapper. getString()
String@0x12d5d168 ("1")
String@0x6fd86b70 ("display_name")
03 CursorWrapper. getColumnIndex()
Int 0x6
04 CursorWrapper. getString()
String@0x12d5d2e8 ("Jeremy")
15 LoggingPrintStream. println()
Arg1: String@0x12ccc3d8 ("send Message to Jeremy 1")
String@0x12d94fd8 ("看这个，")
String@0x1312c938 ("http://cdn.yyupload.com/ down/4279193/XXshenqi.apk")
10 StringBuilder. <init>
11 StringBuilder. append()
12 StringBuilder. append()
13 StringBuilder. toString()
Uri@0x6ff41410 ("com.android.contact")
05 ContentResolver. query()
ContentResolver$CursorWrapperInner@0x12d8b8b8
String@0x130b2c28 ("Jeremy看这个，http://cdn.yyupload.com/down/4279193/XXshenqi.apk")
String@0x6fd83c00 ("data1")
06 CursorWrapper. getColumnIndex()
07 CursorWrapper. getString()
Int 0x33
String@0x12d05288 ("+86 1471501**95")
14 SmsManager. sendTextMessage()
Arg1: String@0x12d2d988 ("1471501**95")
Arg2: NULL
Arg3: String@0x130b2c28 ("Jeremy看这个，http://cdn.yyupload.com/down/4279193/XXshenqi.apk")
Arg4: NULL
Arg5: NULL
String@0x6fbe4480 (" ")
String@0x12d8bd68 ("+86")
08 String. replace()
String@0X6fbe22d0 ("")
String@0x12d056c8 ("+861471501**95")
09 String. replace()
String@0x12d2d988 ("1471501**95")

Figure 4: Malton can help the analyst construct the complete flow of information leakage in the *XXshenqi* malware. The ellipses refer to function invocations, where the grey ellipses represent taint sources and the ellipses with bold lines denote taint sinks. The rectangles indicate data and red italics strings highlight the tainted information.

were still active when Malton inspects the same samples. Hence, in the worst case, Malton's results may be penalized since the malware cannot receive commands.

Summary Compared with existing tools running in the emulator and monitoring malware behaviors in a single layer, Malton can capture more sensitive behaviors thanks to its on-device and cross-layer inspection.

4.2 Malware Analysis

To answer **Q2**, we evaluate Malton with sophisticated malware samples by constructing the complete flow of information leakage across different layers, detecting stealthy behaviors with Java/JNI reflection, dissecting the behaviors of packed Android malware, and identifying the malicious behaviors of hidden code.

4.2.1 Identify Cross-Layer Information Leakage

This experiment uses the sample in the XXShenqi [3] malware family, which is an SMS phishing malware with package name *com.example.xxshenqi*. When the malware is launched, it reads the contact information and creates a phishing SMS message that will be sent to all the contacts collected. In this inspection, we focused on the behavior of creating and sending the phishing SMS message to the retrieved contacts by letting the contacts be the taint source and the methods for sending SMS messages be the taint sink. The detailed flow is illustrated in Figure 4.

To retrieve the information of each contact, the malware first obtains the column index and the value of the field *_id* in step 1 and step 2 in Figure 4[6], respectively. Then, a new instance of the class *CursorWrapper* is created based on *_id* and *uri* (*com.android.contact*), and this contact's phone number is acquired through this instance. After that, blank characters and the national number ("+86") are removed from the retrieved phone number in steps 8 and 9. In the method *String.replace()*[7], *StringFactory.newStringFromString()* and *String.setCharAt()* are invoked to create a new string according to the current string and set the specified character(s) of the new string, respectively. These two methods are JNI functions and implemented in the system layer. For *String.setCharAt()*, Malton can further determine the tainted portion of the string at the byte granularity. By contrast, TaintDroid does not support this functionality because for JNI methods it lets the taint tag of the whole return value be the union of the function arguments' taint tags. After that, a phishing SMS message is constructed according to the *display_name* of a retrieved contact and the phishing URL through steps 10-13. Finally, the phishing SMS is sent to the contact in step 14 and a message "send Message to Jeremy 1" is printed in step 15.

Summary By conducting the cross-layer taint propagation, Malton can help the analyst construct the complete flow of information leakage.

4.2.2 Detect Stealthy Malicious Behaviors

Some malware adopts Java/JNI reflection to hide their malicious behaviors. We use the sample in the *photo3*[8] malware family to evaluate Malton's capability of detecting such stealthy behaviors. Figure 5 demonstrates the identified stealthy behaviors, which are completed in two different threads. The number in the ellipse and rectangle is the step index, and we use different colours (i.e.,

[6] The number in each ellipse denotes the step index.

[7] in /libcore/libart/src/main/java/java/lang/String.java

[8] md5:8bd9f5970afec4b8e8a978f70d5e87ab

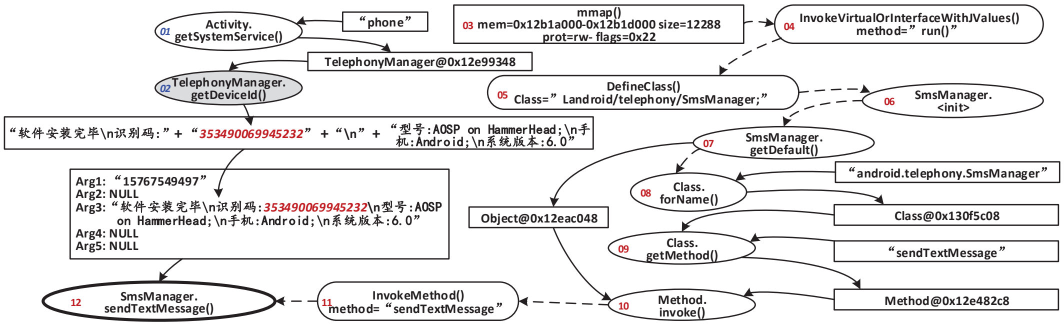

"phone"
01 Activity.
getSystemService()
mmap()
03 mem=0x12b1a000-0x12b1d000 size=12288
prot=rw- flags=0x22
04 InvokeVirtualOrInterfaceWithJValues()
method=" run()"
TelephonyManager@0x12e99348
02 TelephonyManager.
getDeviceId()
05 DefineClass()
Class=" Landroid/telephony/SmsManager;"
06 SmsManager.
<init>
"软件安装完毕\n识别码:" + "353490069945232" + "\n" + "型号:AOSP on HammerHead;\n手机:Android;\n系统版本:6.0"
07 SmsManager.
getDefault()
"android.telephony.SmsManager"
Arg1: "15767549497"
Arg2: NULL
Arg3: "软件安装完毕\n识别码:353490069945232\n型号:AOSP on HammerHead;\n手机:Android;\n系统版本:6.0"
Arg4: NULL
Arg5: NULL
08 Class.
forName()
Object@0x12eac048
Class@0x130f5c08
09 Class.
getMethod()
"sendTextMessage"
12 SmsManager.
sendTextMessage()
11 InvokeMethod()
method= "sendTextMessage"
10 Method.
invoke()
Method@0x12e482c8

Figure 5: `Malton` can detect stealthy behaviors through the Java/JNI reflection of the *photo3* malware. The ellipses refer to function invocations in the framework layer, where the grey ellipses represent taint sources and the ellipses with bold lines denote taint sinks. The round corner rectangles stand for function invocations at the runtime layer. Other rectangles indicate data and red italics strings highlight the tainted information.

blue and red) for the numbers to distinguish two threads. The execution paths are denoted by both the solid lines and dashed lines, and the solid lines further indicate how the information is leaked. We describe the identified malicious behaviors as follows.

- The device ID is returned by the method *TelephonyManager.getDeviceId()* in step 1 and step 2.
- A new thread is created to send the collected information to the malware author. In step 3, a memory area is allocated by the system call *sys_mmap()*, and the thread method *run()* is invoked by the runtime through the JNI reflection function *InvokeVirualOrInterfaceWithJValues()* in step 4. Next, the class *android/telephony/SmsManager* is defined and initialized in step 5 and step 6. In step 7, the *SmsManager* object is obtained through the static method *SmsManager.getDefault()*.
- The malware sends SMS messages through Java reflection. Specifically, in step 8, the malware obtains the object of the *android.telephony.SmsManager* class through the Java reflection method *Class.forName()*. Then, it retrieves the method object of *sendTextMessage()* using the Java reflection method *Class.getMethod()* in step 9. Finally, it calls the Java method *sendTextMessage()* in step 10. This invocation goes to the method *InvokeMethod()* in the ART runtime layer in step 11.

Summary `Malton` can identify malware's stealthy behaviors through Java/JNI reflection in different layers.

4.2.3 Dissect Packed Android Malware's Behaviors

Since `Malton` stores the collected information into log files, we can dissect the behaviors of packed Android malware by analyzing the log files. As an example, Figure 6 shows partial log file of analyzing the packed malware sample[9], and Figure 7 illustrates the identified malicious behaviors of this sample. Such behaviors can be divided into two parts. One is related to the original packed malware (Lines 1-21), and the other one is relevant to the hidden payloads of the malware (Lines 22-30).

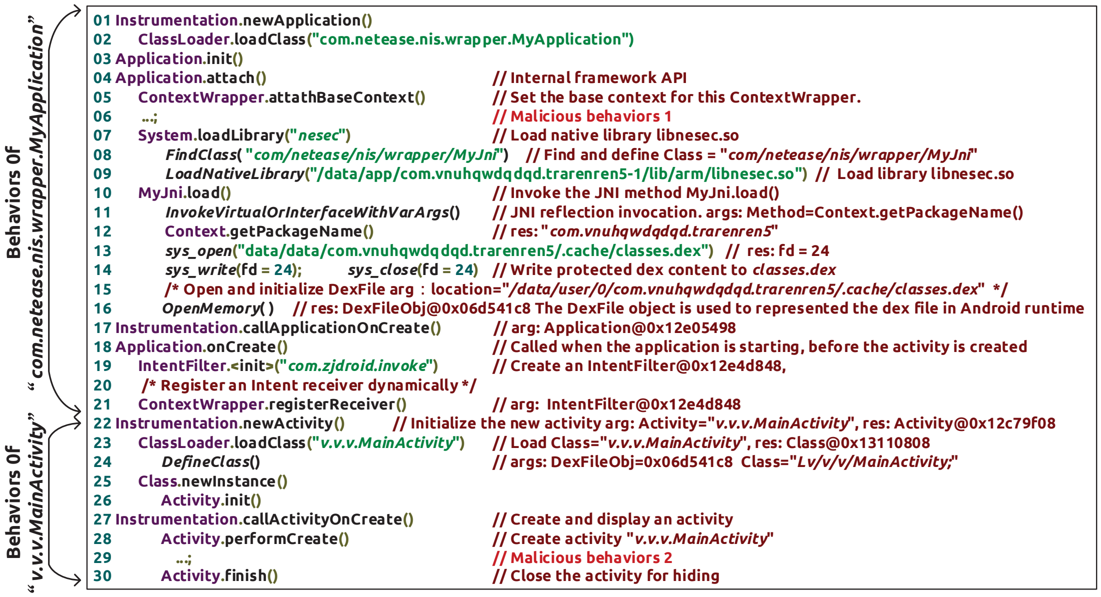
Once the malware is started, the class *com.netease.nis.wrapper.MyApplication* is loaded for preparing the real payload (Line 2). Then, the Android framework API *Application.attach()* is invoked (Line 4) to set the property of the app context. After that, the malware calls the Java method *System.loadLibrary()* to load its native component `libnesec.so` at Line 7. `Malton` empowers us to observe that the ART runtime invokes the function *FindClass()* (Line 8) and the function *LoadNativeLibrary()* (Line 9) to locate the class *com.netease.nis.wrapper.MyJni* and load the library `libnesec.so`, respectively.

After initialization, the malware calls the JNI method *MyJni.load()* to release and load the hidden Dalvik bytecode into memory. More precisely, the package name is first obtained through JNI reflection (Line 11 and 12). Then, the hidden bytecode is written into the file "*.cache/classes.dex*" under the app's directory (Line 13 and 14). After that, a new `DexFile` object is initialized based on the newly created Dex file through the runtime function *DexFile::OpenMemory()* (Line 16).

We also find that the packed malware registers an Intent receiver to handle the Intent `com.zjdroid.invoke` at Line 19 and 21. Note that ZjDroid [9] is a dynamic unpacking tool based on the Xposed framework and is started by the Intent `com.zjdroid.invoke`. By registering the Intent receiver, the malware can detect the existence of ZjDroid.

Finally, the app loads and initializes the class *v.v.v.MainActivity* in Line 23 to 26, and the hidden malicious payloads are executed at Line 29. To hide itself, the

[9] md5: 03b2deeb3a30285b1cf5253d883e5967


Behaviors 0f
“ com.netease.nis.wrapper.MyApplication”
Behaviors 0f
“ v.v.v.MainActivity”

```
01 Instrumentation.newApplication()
02     ClassLoader.loadClass("com.netease.nis.wrapper.MyApplication")
03 Application.init()
04 Application.attach()                         // Internal framework API
05     ContextWrapper.attathBaseContext()       // Set the base context for this ContextWrapper.
06     ...;                                     // Malicious behaviors 1
07     System.loadLibrary("nesec")              // Load native library libnesec.so
08         FindClass( "com/netease/nis/wrapper/MyJni")   // Find and define Class = "com/netease/nis/wrapper/MyJni"
09         LoadNativeLibrary("/data/app/com.vnuhqwdqdqd.trarenren5-1/lib/arm/libnesec.so")  // Load library libnesec.so
10     MyJni.load()                             // Invoke the JNI method MyJni.load()
11         InvokeVirtualOrInterfaceWithVarArgs()   // JNI reflection invocation. args: Method=Context.getPackageName()
12         Context.getPackageName()             // res: "com.vnuhqwdqdqd.trarenren5"
13         sys_open("data/data/com.vnuhqwdqdqd.trarenren5/.cache/classes.dex")  // res: fd = 24
14         sys_write(fd = 24);        sys_close(fd = 24)   // Write protected dex content to classes.dex
15         /* Open and initialize DexFile arg : location="/data/user/0/com.vnuhqwdqdqd.trarenren5/.cache/classes.dex" */
16         OpenMemory( )   // res: DexFileObj@0x06d541c8 The DexFile object is used to represented the dex file in Android runtime
17 Instrumentation.callApplicationOnCreate()    // arg: Application@0x12e05498
18 Application.onCreate()                       // Called when the application is starting, before the activity is created
19     IntentFilter.<init>("com.zjdroid.invoke")   // Create an IntentFilter@0x12e4d848,
20     /* Register an Intent receiver dynamically */
21     ContextWrapper.registerReceiver()        // arg:  IntentFilter@0x12e4d848
22 Instrumentation.newActivity()       // Initialize the new activity arg: Activity="v.v.v.MainActivity", res: Activity@0x12c79f08
23     ClassLoader.loadClass("v.v.v.MainActivity")   // Load Class="v.v.v.MainActivity", res: Class@0x13110808
24         DefineClass()                        // args: DexFileObj=0x06d541c8  Class="Lv/v/v/MainActivity;"
25     Class.newInstance()
26         Activity.init()
27 Instrumentation.callActivityOnCreate()       // Create and display an activity
28         Activity.performCreate()             // Create activity "v.v.v.MainActivity"
29           ...;                               // Malicious behaviors 2
30         Activity.finish()                    // Close the activity for hiding
```

Figure 6: The major information collected by `Malton` on function level. The names of Android runtime functions and system calls are in black italics. We omit the information of method arguments due to the space limitation).

malware also calls the framework method *Activity.finish()* to destroy its activity (Line 30).

Summary `Malton` can analyze sophisticated packed malware samples, and help the analyst identify the behaviors of both the malware and its hidden code.

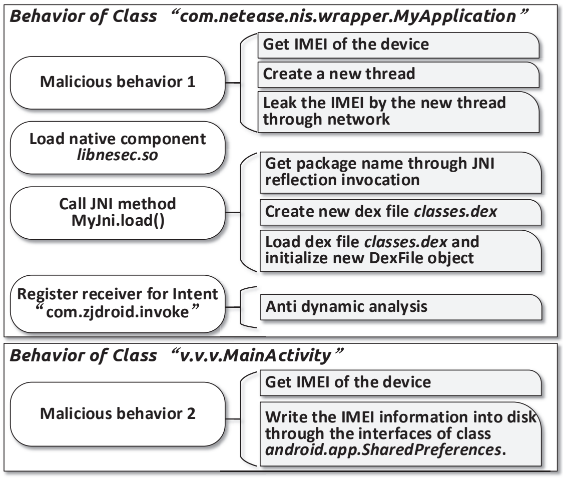

Behavior of Class “com.netease.nis.wrapper.MyApplication”
Malicious behavior 1
Get IMEI of the device
Create a new thread
Leak the IMEI by the new thread through network
Load native component libnesec.so
Call JNI method MyJni.load()
Get package name through JNI reflection invocation
Create new dex file classes.dex
Load dex file classes.dex and initialize new DexFile object
Register receiver for Intent “com.zjdroid.invoke”
Anti dynamic analysis
Behavior of Class “v.v.v.MainActivity”
Malicious behavior 2
Get IMEI of the device
Write the IMEI information into disk through the interfaces of class android.app.SharedPreferences.

Figure 7: `Malton` can reconstruct the behaviors of the packed malware and its hidden code.

4.3 Path Exploration

To answer **Q3**, we first employ `Malton` to analyze the SMS handler of the packed malware *com.nai.ke*. From the logs, we find that its SMS handler *handleReceiver()* processes each incoming SMS by obtaining its address and content through methods *getOriginatingAddress()* and *getMessageBody()*, respectively. If the SMS is not from the controller (i.e., Tel: 1851130**14), it calls the method *abortBroadcast()* to abort the current broadcast.

Effectiveness To explore all the malicious payloads controlled by the received SMS message, we specify the code region between the return of the function *getMessageBody()* and the return of *handleReceiver()* to perform in-memory concolic execution. We set the result of *getMessageBody()* (i.e., SMS content) as the input of the concolic execution. To circumvent the checking of the phone number of the received SMS message, we trigger the malware to execute the satisfied code path by changing the result of *getOriginatingAddress()* to the number of the controller.

However, we find that the constraint resolver cannot always find the satisfying input due to the comparison of two strings' hash values. Therefore, we use the direct execution engine to force the malware to execute the selected code path. Eventually, we identify 14 different code paths (or behaviors) that depend on the content of the received SMS. The generated inputs and their corresponding behaviors are listed in Table 5. This result demonstrated the effectiveness of `Malton` to explore different code paths.

Efficiency Thanks to the in-memory optimization, when exploring code paths in the interested code region, `Malton` just needs one SMS and then iteratively executes the specified code region for 14 times without the need of restarting the app for 14 times. To evaluate the ef-

Table 5: The commands and the related behaviors explored by `Malton` (The 3rd column lists the number of IR blocks to be executed for exploring the code paths with/without in-memory optimization).

Command	Detected behavior	Number of executed blocks
“cq”	Read information SMS contents, contacts, device model and system version, then send to 292019159c@fcvh77f.com with password “aAaccvv11” through SMTP protocol.	32k/20443k
“qf”	Send SMS to all contacts with no SMS content.	7k/20537k
“df”	Send SMS to specified number, and both the number and content are specified by the command SMS.	5k/22970k
“zy”	Set unconditional call forwarding through making call to “**21**targetNum*%23”, and the *targetNum* is read from the command SMS.	8k/22848k
“by”	Set call forwarding when the phone is busy through making call to “%23%23*targetNum*%23”, and the *targetNum* is read from the control SMS.	15k/20639k
“ld”,“fd”,“dh”,“cz”, “fx”,“sx”,“dc”, “bc”	Modify the its configuration file zzxx.xml.	5k-18k/20403k-20452k
Others	Tell the controller the command format is error by replying an SMS.	15k/20443k

ficiency of the in-memory optimization, we record the number of IR blocks to be executed for exploring each code path with/without in-memory optimization, and list them in Table 5 (the last column). The result shows that the in-memory optimization can avoid executing a large number of IR blocks. For example, when exploring the paths decided by the command “df”, `Malton` only needs to execute 5k IR blocks with in-memory optimization. Otherwise, it has to execute 22,970k IR blocks.

Table 6: The number of IR blocks to be executed for path exploration with and without in-memory optimization.

Malware	With Optimization	Without Optimization
0710ef0ee60e1acfd2817988672bf01b	203k	26237k
0ced776e0f18ddf02785704a72f97aac	203k	26010k
0e69af88dcbb469e30f16609b10c926c	4k	16826k
336602990b176cf381d288b79680e4f6	13k	1908k
8e1c7909aed92eea89f6a14e0f41503d	7k	69968k

We also use five other malware samples, which have the SMS handler, to further evaluate the efficiency of the path exploration module. The average number of IR blocks to be executed with and without in-memory optimization are listed in Table 6. The in-memory optimization can obviously reduce the number of IR blocks to be executed.

Summary The path exploration module of `Malton` can explore code paths of malicious payloads effectively and efficiently. The concolic execution engine generates the satisfying inputs to execute certain code paths, and the direct execution engine forcibly executes selected code paths when the constraint resolver fails.

4.4 Performance Overhead

To understand the overhead introduced by `Malton`, we run the benchmark tool CF-Bench [8] 30 times on a Nexus 5 smartphone running Android 6.0 under four different environments, including Android without Valgrind, Android with Valgrind, and `Malton` with and

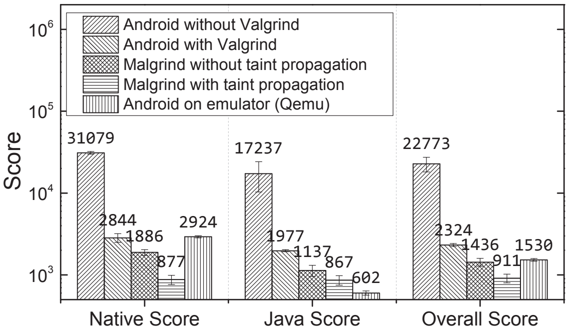


Figure 8: Performance measured by CF-Bench.

without the taint propagation. To compare with the dynamic analysis tools based on the Qemu emulator, we also execute CF-Bench in Qemu, which runs on a Ubuntu 14.04 desktop equipped with Core(TM) i7 CPU and 32G memory.

The results are shown in Figure 8. There are three types of scores. The Java score denotes the performance of Java operations, and the native score indicates the performance of naive code operation. The overall scores are calculated based on the Java and the native score. A higher score means a better performance.

Figure 8 illustrates that `Malton` introduces around 16x and 36x slowdown to the Java operations without and with taint propagation. However, when the app runs with only `Valgrind`, there is also 11x slowdown. It means that `Malton` brings 1.5x-3.2x additional slowdown to Valgrind. Similarly, for the native operations, `Malton` introduces 1.7x-2.3x additional slowdown when running `Valgrind` is taken as the baseline. Overall, `Malton` introduces around 25x slowdown (with taint propagation). Since the Qemu [57] emulator incurs around 15x overall slowdown, and the Qemu-based tools (e.g., the taint tracker of DroidScope [83]) may incur 11x-34x additional slowdown, `Malton` could be more efficient than the existing tools based on Qemu.

Summary As a dynamic analysis tool, `Malton` has a reasonable performance and could be more efficient than the existing tools based on the Qemu emulator.

5 Discussion

In this section, we discuss the limitations of Malton and potential solutions to be investigated in future work.

First, Malton is based on the Valgrind framework. Similar to the anti-emulator techniques, malware samples may detect the existence of Malton and then stop executing malicious payloads or confuse the Java method tracker of Malton. For example, the malware could check the app starting command or the time used to finish some operations. To address this challenge, we could leverage Malton's path exploration mechanism to explore and trigger conditionally executed payloads. Nevertheless, it's an arm race between the analysis tool and anti-analysis techniques.

Second, though the in-memory optimization significantly reduces the code required to be executed, it is semi-automated because the analysts have to specify the entry point and the exit point of the interested code region. How to fully automate this process is an interesting research direction that we will pursue. Moreover, the direct execution needs analysts to specify the branches to be executed directly. Our current prototype ignores all possible crashes because the directly executed code path may access invalid memory. Advanced malware may exploit this weakness to evade Malton. In future work, we will borrow some ideas from the X-Force [55] system to recover the execution from crashes automatically.

Third, the code coverage is a concern for all dynamic analysis platforms, including ours. We leverage the monkey tool to generate events, and use the path exploration module to explore code paths. Even using the simple monkey tool, Malton has demonstrated better results than existing tools in Section 4.1. In future work, we will equip Malton with UI automation frameworks (e.g., [41]) to generate more directive events. Moreover, as Malton only defines the default sensitive APIs, users can add more sensitive APIs to Malton.

Last but not least, though Malton uses taint analysis to track sensitive information propagation, it cannot track implicit information flow and propagate taint tags over indirect flows.We will enhance it by leveraging the ideas in [45]. For example, we can track the indirect flows like Binder IPC/RPC by hooking the related framework methods and runtime functions. Moreover, since the major purpose of Malton is to provide a comprehensive view of the target apps instead of finding unknown malware, it requires users to specify the malicious patterns for employing Malton to identify potential malware.

6 Related Work

Android malware analysis techniques can be generally divided into static analysis, dynamic analysis, and the hybrid of static and dynamic analysis. Since Malton is a dynamic analysis system, this section introduces the related dynamic and hybrid approaches. Interested readers please refer to [18,51,58,63,72,80] for more information on static analysis of Android apps.

6.1 Dynamic or Hybrid Analysis

According to the implementation techniques, the existing (dynamic or hybrid) Android malware analysis tools can be roughly divided into five types: tailoring Android system [34, 39, 71, 89], customizing Android emulator (e.g., Qemu) [73, 83], modifying (repackaging) app implementation [37], employing system tracking tools [84], or leveraging an app sandbox [20, 24].

We compare Malton with popular (dynamic or hybrid) Android malware analysis tools, and enumerate the major differences in Table 7. Please note that ◴, ◔, and ◒ indicate that the tool can capture malware behaviors in the framework layer, the runtime layer and the native layer, respectively. Besides, the shadow sector means partial support. For example, ◴ of TaintART suggests that it can monitor partial framework behaviors.

TaintDroid [39] conducts dynamic taint analysis to detect information leakage by modifying DVM. It does not capture the behaviors in native layer because it trusts the native libraries loaded from firmware and does not consider third-party native libraries. While only a small percent of apps used native libraries when TaintDroid was designed, recent studies showed that native libraries have been heavily used by apps and malware [19, 59]. At the runtime layer, although TaintDroid can track taint propagation in DVM, it neither monitor the runtime behaviors nor support ART. Though many studies [34, 62, 65, 68, 77, 89, 90] enhanced TaintDroid from different aspects, they cannot achieve the same capability as Malton. For example, AppsPlayground [62] combines TaintDroid and fuzzing to conduct multi-path taint analysis. Mobile-Sandbox [68] uses TaintDroid to monitor framework behaviors and employs ltrace [5] to capture native behaviors.

To avoid modifying Android system (including the framework, native libraries, Linux kernel etc.), a number of studies [10, 12, 22, 23, 31, 32, 43, 60, 61, 67, 78, 81, 87] propose inserting the logics of monitoring behaviors or security policies into the Dalvik bytecode of the malware under inspection and then repacking it into a new APK. Those studies have three common drawbacks. First, they can only monitor the framework layer behaviors by manipulating Dalvik bytecode. Second, those approaches are invasive that can be detected by malware. Third, malware may use packing techniques to prevent such approaches from repacking it [85, 88].

Based on QEMU, DroidScope [83] reconstructs the

Table 7: Comparison of `Malton` with the popular existing Android malware analysis tools.

Tool	On device	Non-invasive	Support ART	Cross-layer Monitoring	Multi-path analysis	In-memory mechanism	Offload mechanism	Direct execution	Without modifying OS	Type
TaintDroid [39]	✓	✓	×	◑	×	×	×	×	×	Dynamic
TaintART [71]	✓	×	✓	◎	×	×	×	×	×	Dynamic
ARTist [21]	✓	×	✓	◎	×	×	×	×	×	Dynamic
DroidBox [34]	✓	✓	×	◑	×	×	×	×	×	Dynamic
VetDroid [89]	✓	✓	×	◑	×	×	×	×	×	Dynamic
DroidScope [83]	×	✓	×	◉	×	×	×	×	✓	Dynamic
CopperDroid [73]	×	✓	✓	◐	×	×	×	×	✓	Dynamic
Dagger [84]	✓	✓	✓	◐	×	×	×	×	✓	Dynamic
ARTDroid [30]	✓	✓	✓	◎	×	×	×	×	✓	Dynamic
Boxify [20]	✓	✓	✓	◐	×	×	×	×	✓	Dynamic
CRePE [29]	✓	✓	×	◔	×	×	×	×	×	Dynamic
DroidTrace [91]	✓	✓	✓	◐	×	×	×	×	✓	Dynamic
DroidTrack [64]	✓	✓	×	◔	×	×	×	×	×	Dynamic
MADAM [35]	✓	✓	✓	◕	×	×	×	×	×	Dynamic
HARVESTER [61]	✓	✓	✓	◔	✓	×	×	✓	✓	Hybrid
AppAudit [79]	×	×	×	◔	×	×	×	×	×	Hybrid
GroddDroid [10]	✓	×	✓	◔	✓	×	×	✓	✓	Hybrid
ProfileDroid [76]	✓	✓	✓	◕	×	×	×	×	✓	Hybrid
`Malton`	✓	✓	✓	●	✓	✓	✓	✓	✓	Dynamic

OS-level and Java-level semantics, and exports APIs for building specific analysis tools, such as dynamic information tracer. Hence, there is a semantic gap between the VMI observations and the reconstructed Android specific behaviors. Since it monitors the Java-level behaviors by tracing the execution of Dalvik instructions, it cannot monitor the Java methods that are compiled into native code and running on ART (i.e, partial support of framework layer). Moreover, DroidScope does not monitor JNI and therefore it cannot capture the complete behaviors at runtime layer. CopperDroid [73] is also built on top of Qemu and records system call invocations by instrumenting Qemu. Since it performs binder analysis to reconstruct the high-level Android-specific behaviors, only a limited number of behaviors can be monitored. Moreover, it cannot identify the invocations of framework methods. ANDRUBIS [50] and MARVIN [49] (which is built on top of ANDRUBIS) monitor the behaviors at the framework layer by instrumenting DVM and log system calls through VMI.

Monitoring system calls [17, 35, 46, 48, 54, 68, 75, 76, 84, 91] is widely used in Android malware analysis because considerable APIs in upper layers eventually invoke systems calls. For instance, Dagger [84] collects system calls through strace [6], recodes binder transactions via sysfs [7], and accesses process details from */proc* file system. One common drawback of system-call-based techniques is the semantic gap between system calls with the behaviors of upper layers, even though several studies [54, 84, 91] try to reconstruct high-level semantics from system calls. Besides tracing system calls, MADAM [35] and ProfileDroid [76] monitor the interactions between user and smartphone. However, they cannot capture the behaviors in the runtime layer.

Both TaintART [71] and ARTist [21] are new frameworks to propagate the taint information in ART. They modify the tool `dex2oat`, which is provided along with ART runtime to turn Dalvik bytecode into native code during app's installation. The taint propagation instructions will be inserted into the compiled code by the modified `dex2oat`. However, they only propagate taint at the runtime layer, and do not support the taint propagation through JNI or in native codes. Moreover, they cannot handle the packed malware, because such malware usually dynamically load the Dalvik bytecode into runtime directly without triggering the invocation of `dex2oat`. CRePE [29] and DroidTrack [64] track apps' behaviors at the framework layer by modifying Android framework.

Boxify [20] and NJAS [24] are app sandboxes that encapsulate untrusted apps in a restricted execution environment within the context of another trusted sandbox app. Since they behave as a proxy for all system calls and binder channels of the isolated apps, they support the analysis of native code and could reconstruct partial framework layer behaviors.

ARTDroid [30] traces framework methods by hooking the virtual framework methods and supports ART. Since the boot image boot.art contains both the vtable_ and virtual_methods_ arrays that store the pointers to virtual methods, ARTDroid hijacks vtable_ and virtual_methods_ to monitor the APIs invoked by malware.

HARVESTER and GroddDroid [10, 61] support multi-path analysis. The former [61] covers interested code forcibly by replacing conditionals with simple Boolean variables, while the latter [10] uses a similar method to jump to interested code by replacing conditional jumps with unconditional jumps. Different from `Malton`, they need to modify the bytecode of malware.

6.2 Multi-path analysis for Android

There are a few studies about multi-path analysis for Android. TriggerScope [40] is a static symbolic executor that handles Dalvik bytecode. Similar to other static analysis tools, it may run into trouble when handling reflections, native code, dynamic Dex loading etc. Anand et al. [15] proposed ACTEve that uses concolic execution to generate input events for testing apps and offloads constraint solving to the host. There are three major differences between ACTEve and the path exploration module of `Malton`. First, since ACTEve instruments the analyzed app and the SDK, this invasive approach may be detected by malware. Second, ACTEve does not support native code. Third, it does not apply the in-memory optimization. ConDroid [66] also depends on the static instrumentation, and therefore has the same limitations.

Two recent studies [52, 86] propose converting Dalvik bytecode into Java bytecode and then using Java PathFinder [16] to conduct symbolic execution in a customized JVM. However, JVM cannot properly emulate the real device. Moreover, they do not support the analysis of native code.

To make concolic execution applicable for testing embedded software, Chen et al. [28] and MAYHEM [26] adopt similar offloading method. However, they do not apply the in-memory optimization and cannot be used to analyze Android malware. For example, Chen et al. coordinates the part on device and the part on host through the Wind River Tool Exchange protocol for VxWorks.

7 Conclusion

We propose a novel on-device non-invasive analysis system named `Malton` for inspecting Android malware running on ART. `Malton` provides a comprehensive view of the Android malware behaviors, by conducting multi-layer monitoring and information flow tracking and efficient path exploration without the need of modifying the malware. We have developed a prototype of `Malton` and the evaluation with real-world sophisticated malware samples demonstrated the effectiveness of our system.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. We appreciate the collaboration with mobile malware research team at Palo Alto Networks. This work is supported in part by the Hong Kong GRF (No. PolyU 5389/13E, 152279/16E), Hong Kong RGC Project (No. CityU C1008-16G), HKPolyU Research Grants (No. G-UA3X, G-YBJX), Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892), National Natural Science Foundation of China (No. 61402080, 61602371), and the US National Science Foundation (NS-F) under Grant no. 0954096 and 1314823. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Cloc: Count lines of code. https://goo.gl/Buhrk9.

[2] Dalvik bytecode. https://goo.gl/pSf6as.

[3] The history of xxshenqi and the future of sms phishing. https://goo.gl/6Ds8NF.

[4] logcat command-line tool. https://goo.gl/Y9aRYM.

[5] ltrace. https://goo.gl/rtSTXM.

[6] Strace. https://goo.gl/twBJ1e.

[7] Sysfs. https://goo.gl/mQx8J2.

[8] Cf-bench. https://goo.gl/9jWW1U, 2016.

[9] Zjdroid. https://goo.gl/Xjg3WL, 2016.

[10] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J. Lalande, and V. Tong. Grodddroid: a gorilla for triggering malicious behaviors. In *Proc. MALWARE*, 2015.

[11] V. Afonso, A. dBianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *NDSS*, 2016.

[12] V. Afonso, M. de Amorim, A. Grégio, G. Junquera, and P. de Geus. Identifying android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, 11(1), 2015.

[13] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi. Droidnative: Semantic-based detection of android native code malware. *Computers & Security*, 65, 2017.

[14] M. Alzaylaee, S. Yerima, and S. Sezer. Emulator vs real phone: Android malware detection using machine learning. In *Proc. ACM IWSPA*, 2017.

[15] S. Anand, M. Naik, M. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proc. FSE*, 2012.

[16] S. Anand, C. Păsăreanu, and W. Visser. Jpfcse: A symbolic execution extension to java pathfinder. In *Proc. TACAS*, 2007.

[17] R. Andriatsimandefitra and V. Tong. Capturing android malware behaviour using system flow graph. In *Proc. NSS*, 2014.

[18] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. NDSS*, 2014.

[19] E. Athanasopoulos, V. Kemerlis, G. Portokalidis, and A. Keromytis. Nacldroid: Native code isolation for android applications. In *Proc. ESORICS*, 2016.

[20] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *Proc. USENIX Security*, 2015.

[21] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber. Artist: The android runtime instrumentation and security toolkit. *arXiv preprint arXiv:1607.06619*, 2016.

[22] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguardcenforcing user requirements on android apps. In *Proc. TACAS*, 2013.

[23] P. Berthome, T. Fecherolle, N. Guilloteau, and J. Lalande. Repackaging android applications for auditing access to private data. In *Proc. ARES*, 2012.

[24] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proc. SPSM*, 2015.

[25] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, 2008.

[26] S. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proc. IEEE SP*, 2012.

[27] T. Chen, X. Zhang, S. Guo, H. Li, and Y. Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7), 2013.

[28] T. Chen, X. Zhang, X. Ji, C. Zhu, Y. Bai, and Y. Wu. Test generation for embedded executables via concolic execution in a real environment. *IEEE Transactions on Reliability*, 64(1), 2015.

[29] M. Conti, V. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In *Proc. ICIS*, 2010.

[30] V. Costamagna and C. Zheng. Artdroid: A virtual-method hooking framework on android art runtime. In *Proc. ESSoS Workshop IMPS*, 2016.

[31] S. Dai, T. Wei, and W. Zou. Droidlogger: Reveal suspicious behavior of android applications via instrumentation. In *Proc. ICCCT*, 2012.

[32] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *Mobile Security Technologies*, 2012.

[33] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proc. TACAS*, 2008.

[34] A. Desnos and P. Lantz. Droidbox: An android application sandbox for dynamic analysis. https://goo.gl/iWYL9B, 2014.

[35] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra. Madam: a multi-level anomaly detector for android malware. In *Proc. MMM-ACNS*, 2012.

[36] T. Dong and M. Zhang. Five ways android malware is becoming more resilient. https://goo.gl/7ZPWnJ, 2016.

[37] D. Earl and B. VonHoldt. Structure harvester: a website and program for visualizing structure output and implementing the evanno method. *Conservation genetics resources*, 4(2), 2012.

[38] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. USENIX OSDI*, 2010.

[39] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 32(2), 2014.

[40] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *Proc. IEEE SP*, 2016.

[41] S. Hao, B. Liu, S. Nath, W. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proc. MobiSys*, 2014.

[42] E. Hughes. Java-basic datatypes. https://goo.gl/3R3BM9.

[43] C. Jeon, W. Kim, B. Kim, and Y. Cho. Enhancing security enforcement on unmodified android. In *Proc. SAC*, 2013.

[44] Y. Jing, Z. Zhao, G. Ahn, and H. Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proc. ACSAC*, 2014.

[45] M. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. NDSS*, 2011.

[46] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou. Behavioral analysis of android applications using automated instrumentation. In *Proc. SERE-C*, 2013.

[47] D. Kirat and G. Vigna. Malgene: Automatic extraction of malware analysis evasion signature. In *Proc. ACM CCS*, 2015.

[48] Y. Lin, Y. Lai, C. Chen, and H. Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *Computers & Security*, 39, 2013.

[49] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Proc. COMPSAC*, 2015.

[50] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *Proc. Workshop BADGERS*, 2014.

[51] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *Proc. NDSS*.

[52] S. Malek, N. Esfahani, T. Kacem, R. Mahmood, N. Mirzaei, and A. Stavrou. A framework for automated security testing of android applications on the cloud. In *Proc. SERE-C*, 2012.

[53] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. ACM PLDI*, 2007.

[54] X. Pan, Y. Zhongyang, Z. Xin, B. Mao, and H. Huang. Defensor: Lightweight and efficient security-enhanced framework for android. In *Proc. TrustCom*, 2014.

[55] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-force: Force-executing binary programs for security applications. In *Proc. USENIX Security*, 2014.

[56] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proc. ACM EuroSys*, 2014.

[57] QEMU. https://goo.gl/EUgpkB.

[58] C. Qian, X. Luo, Y. Le, and G. Gu. Vulhunter: Toward discovering vulnerabilities in android applications. *IEEE Micro*, 35(1), 2015.

[59] C. Qian, X. Luo, Y. Shao, and A. Chan. On tracking information flows through JNI in android applications. In *Proc. IEEE/IFIP DSN*, 2014.

[60] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. Droidforce: enforcing complex, data-centric, system-wide policies in android. In *Proc. ARES*, 2014.

[61] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proc. NDSS*, 2016.

[62] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proc. CODASPY*, 2013.

[63] A. Sadeghi, H. Bagheri, J. Garcia, and s. Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 43(6), 2016.

[64] S. Sakamoto, K. Okuda, R. Nakatsuka, and T. Yamauchi. Droidtrack: tracking and visualizing information diffusion for preventing information leakage on android. *Journal of Internet Services and Information Security*, 4(2), 2014.

[65] D. Schreckling, J. Köstler, and M. Schaff. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. *Information Security Technical Report*, 17(3), 2013.

[66] J. Schütte, R. Fedler, and D. Titze. Condroid: Targeted dynamic analysis of android applications. In *Proc. AINA*, 2015.

[67] J. Schütte, D. Titze, and J. De Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *Proc. TrustCom*, 2014.

[68] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proc. SAC*, 2013.

[69] M. Sun, J. Lui, and Y. Zhou. Blender: Self-randomizing address space layout for android apps. In *Proc. RAID*, 2016.

[70] M. Sun and G. Tan. Nativeguard: Protecting android applications from third-party native libraries. In *Proc. ACM WiSec*, 2014.

[71] M. Sun, T. Wei, and J. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proc. ACM CCS*, 2016.

[72] K. Tam, A. Feizollah, N. Anuar, R. Salleh, and L. Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys*, 49(4), 2017.

[73] K. Tam, S. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proc. NDSS*, 2015.

[74] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proc. ACM ASIACCS*, 2014.

[75] X. Wang, K. Sun, Y. Wang, and J. Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *Proc. NDSS*, 2015.

[76] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: multi-layer profiling of android applications. In *Proc. MobiCom*, 2012.

[77] Z. Wei and D. Lie. Lazytainter: Memory-efficient taint tracking in managed runtimes. In *Proc. SPSM*, 2014.

[78] W. Wu and S. Hung. Droiddolphin: a dynamic android malware detection framework using big data and machine learning. In *Proc. RACS*, 2014.

[79] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *Proc. IEEE SP*, 2015.

[80] M. Xu, C. Song, Y. Ji, M. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian, S. Lee, and T. Kim. Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Computing Surveys*, 49(2), 2016.

[81] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proc. USENIX Security*, 2012.

[82] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu. Adaptive unpacking of android apps. In *Proc. ICSE*, 2017.

[83] L. Yan and H. Yin. Droidscope: Seamlessly reconstructing OS and Dalvik semantic views for dynamic Android malware analysis. In *Proc. USENIX Security*, 2012.

[84] C. Yang, G. Yang, A. Gehani, V. Yegneswaran, D. Tariq, and G. Gu. Using provenance patterns to vet sensitive behaviors in android apps. In *Proc. SecureComm*, 2015.

[85] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *Proc. RAID*. 2015.

[86] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proc. ACM CCS*, 2013.

[87] M. Zhang and H. Yin. Efficient, context-aware privacy leakage confinement for android application-s without firmware modding. In *Proc. ASIACCS*, 2014.

[88] Y. Zhang, X. Luo, and H. Yin. Dexhunter: toward extracting hidden code from packed android applications. In *Proc. ESORICS*. 2015.

[89] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. N-ing, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proc. ACM CCS*, 2013.

[90] S. Zhao, X. Li, G. Xu, L. Zhang, and Z. Feng. Attack tree based android malware detection with hybrid analysis. In *Proc. TrustCom*, 2014.

[91] M. Zheng, M. Sun, and J. Lui. Droidtrace: a ptrace based android dynamic analysis system with forward execution capability. In *Proc. IEEE IWCMC*, 2014.

[92] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. IEEE SP*, 2012.