NCScope: Hardware-Assisted Analyzer for Native Code in Android Apps

Hao Zhou
The Hong Kong Polytechnic University
Hong Kong, China
cshaoz@comp.polyu.edu.hk

Shuohan Wu
The Hong Kong Polytechnic University
Hong Kong, China
tom111.wu@connect.polyu.hk

Xiapu Luo*
The Hong Kong Polytechnic University
Hong Kong, China
csxluo@comp.polyu.edu.hk

Ting Wang
Pennsylvania State University
State College, Pennsylvania, USA
tbw5359@psu.edu

Yajin Zhou
Zhejiang University
Zhejiang, China
yajin_zhou@zju.edu.cn

Chao Zhang
Tsinghua University, BNRist
Beijing, China
chaoz@tsinghua.edu.cn

Haipeng Cai
Washington State University
Pullman, Washington, USA
haipeng.cai@wsu.edu

ABSTRACT

More and more Android apps implement their functionalities in native code, so does malware. Although various approaches have been designed to analyze the native code used by apps, they usually generate incomplete and biased results due to their limitations in obtaining and analyzing high-fidelity execution traces and memory data with low overheads. To fill the gap, in this paper, we propose and develop a novel hardware-assisted analyzer for native code in apps. We leverage ETM, a hardware feature of ARM platform, and eBPF, a kernel component of Android system, to collect real execution traces and relevant memory data of target apps, and design new methods to scrutinize native code according to the collected data. To show the unique capability of `NCScope`, we apply it to four applications that cannot be accomplished by existing tools, including systematic studies on self-protection and anti-analysis mechanisms implemented in native code of apps, analysis of memory corruption in native code, and identification of performance differences between functions in native code. The results uncover that only 26.8% of the analyzed financial apps implement self-protection methods in native code, implying that the security of financial apps is far from expected. Meanwhile, 78.3% of the malicious apps under analysis have anti-analysis behaviors, suggesting that `NCScope` is very useful to malware analysis. Moreover, `NCScope` can effectively detect bugs in native code and identify performance differences.

*The corresponding author.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534410

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Android, Dynamic Analysis, App Analysis

ACM Reference Format:
Hao Zhou, Shuohan Wu, Xiapu Luo, Ting Wang, Yajin Zhou, Chao Zhang, and Haipeng Cai. 2022. NCScope: Hardware-Assisted Analyzer for Native Code in Android Apps. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534410

1 INTRODUCTION

Android continuously dominates the smartphone market with over 70% share [52]. A recent study [90] showed that about 40% benign apps use native code for implementing advanced functionalities or achieving better performance. Sumaya et al. disclosed that native code has been widely used in very popular apps [55]. Many guidelines [28, 37] also suggest that security-sensitive apps (e.g., mobile banking apps) should implement self-protection techniques in native code to prevent them from running in rooted devices and being tampered. Meanwhile, researchers discovered that malicious apps always abuse native code to perform anti-analysis [54, 87, 89, 93, 98, 101]. Therefore, tools for analyzing native code in apps are highly demanded for researchers to characterize apps using native code, for store maintainers to uncover unwanted activities implemented in native code, and for app developers to diagnose bugs and performance issues.

However, it is challenging to analyze native code in Android apps. On the one hand, the complexity of native code [68] and the adopted dynamic features (e.g., dynamically loading) [62, 65, 78, 80] impede the static analysis of native code. On the other hand, although a number of dynamic analysis approaches have been

designed for inspecting native code in apps, they usually generate incomplete and biased results due to their limitations in obtaining and analyzing high-fidelity execution traces and memory data with low overheads. More precisely, the majority of existing dynamic analysis approaches rely on three kinds of underlying techniques, including debugger [99], dynamic binary instrumentation (DBI) frameworks [95], and emulator [54, 87, 92, 93, 98]. Unfortunately, these techniques have the following limitations that negatively affect the analysis of native code.

First, it is difficult for these techniques to collect all executed instructions in an accurate and efficient manner because DBI and emulator may incorrectly recognize and emulate instructions as revealed by recent studies[67, 79, 82] while debuggers can only access an instruction through the single-step method [13]. Consequently, behaviors of native code may be missed due to the incomplete instruction traces. Second, these techniques can be easily defeated by anti-analysis methods in native code[94] because of their obvious footprints. For example, debuggers change status of the debugged process, DBI loads extra libraries and code to the app's memory, and emulator has special files and properties different from real systems (see §4.3). Note that malicious apps will hide their behaviors once they discover the existence of dynamic analysis tools. Third, these underlying techniques introduce significant overhead, which leads to biased profiling results. For instance, DBI and emulator based tools [81, 93, 95, 98] cause more than 10x slowdown. Such a slowdown has been exploited by malware to detect these tools through timing checks [43].

To overcome the aforementioned limitations, we propose a novel hardware-assisted analyzer named `NCScope` for native code in Android apps. Specifically, exploiting ETM (Embedded Trace Microcell), a hardware feature of ARM platform, we collect the executed ARM instructions of the app running in real device instead of emulators without the need of DBI frameworks or software-based debuggers. Note that the overhead of collecting ARM instructions through ETM is very low (around zero [2]). Meanwhile, we use eBPF (Extended Berkeley Packet Filter), a kernel component of Android system introduced since Android 9.0, to efficiently obtain the memory data used by the app. Moreover, we design new methods to identify the behaviors of native code from the collected information by recovering the run-time function calls and data flow. Since `NCScope` achieves efficient instruction tracing and memory data retrieving, the evaluation results (in §5.1) show that it introduces very low additional overhead (1.180x slowdown on average) to the execution of apps, and it can bypass timing checks [43]. Based on the collected data, users can develop analysis methods for characterizing native code in apps.

To demonstrate the unique capability of `NCScope`, we equip it with new analysis methods for four applications (①–④) that cannot be accomplished by existing tools. We first use it to conduct systematic studies on ① self-protection behaviors of financial apps (§5.2) and ② anti-analysis behaviors of malicious apps (§5.3). Then, we use `NCScope` to ③ detect memory corruption caused by CWE-416 (use-after-free) and CWE-415 (double-free) (§5.4), which cannot be diagnosed from the exceptions raised by Android system, and to ④ identify the performance differences between functions used in native code (§5.5).

In summary, we make the following contributions:

- We propose `NCScope`, a new hardware-assisted analyzer for native code in apps. It outperforms existing approaches in terms of effectiveness, efficiency, evasion resilience, and overhead. The source code of `NCScope` is available at https://github.com/moonZHH/NCScope.
- We design new methods for `NCScope` to detect self-protection and anti-analysis methods and diagnose memory corruption in native code by recovering run-time function calls and data flow from the collected instruction traces and memory data.
- We extensively evaluate `NCScope` and use it to conduct systematic studies on the native code based self-protection methods adopted by financial apps and the anti-analysis mechanisms used by malicious apps. The results show that `NCScope` introduces very small additional overhead. Moreover, it discovers that a few (26.8%) of financial apps implement self-protection methods whereas most (78.3%) of malicious apps under examination implement anti-analysis methods in their native code. We also apply `NCScope` to detect memory corruption in native code of apps to show that it can aid bug diagnosis and present a case study to show that `NCScope` can aid performance analysis.

2 BACKGROUND

In this section, we introduce the background knowledge about ETM in §2.1 and eBPF in §2.2.

2.1 Embedded Trace Microcell

Embedded Trace Microcell (ETM) is a hardware feature of ARM platform. It tracks the instructions executed by CPU via monitoring instruction buses with almost no overhead [16]. When tracing the executed instructions of a running app through ETM, it generates *trace elements*, which contain the information (e.g., target addresses of executed branch instructions of the target app) for recovering the app's run-time execution traces, and outputs them as an ETM stream. Among the trace elements, we mainly resolve the *Address* element because it stores the address of each tracked instruction, which will be used to recover the app's run-time behaviors (§4). According to `Futuremark` [8], all of the most popular 50 smartphones are equipped with the ARM processors that support ETM.

2.2 Extended Berkeley Packet Filter

Extended Berkeley Packet Filter (eBPF) is a lightweight and secure in-kernel virtual machine for executing eBPF programs [17]. An eBPF program can hook a kernel instruction by using the kernel probe (kprobe) infrastructure [25] or a kernel tracepoint by using the event tracing infrastructure [47]. Once the target instruction is being executed, the corresponding eBPF program will be executed to perform analyst-specified operations, e.g., getting a register value or retrieving the memory data. Besides supporting the kernel instrumentation, an eBPF program can hook user-space programs through the user-space probe (uprobe) [45]. Since Android 9.0, both the kprobe and uprobe are supported by Android system. As `bcc` [10] provides convenient programming interfaces to customize eBPF programs, we use it to develop eBPF programs.

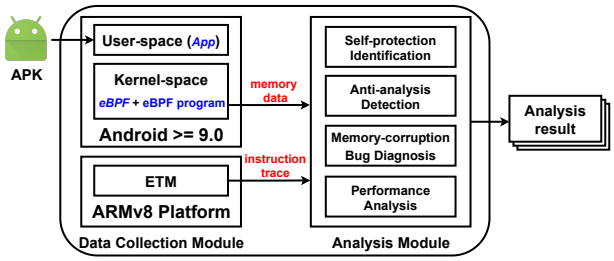

Figure 1: The architecture of NCScope.

3 OVERVIEW OF NCSCOPE

As shown in Figure 1, NCScope consists of two modules: an ARMv8 based data collection module running an Android system (>=9.0) loaded with eBPF programs, and an analysis module implementing various new analysis methods. The rational of this two-module architecture is to minimize the additional overhead. That is, we let ARM platform only perform data collection (i.e., recording the instruction trace and retrieving the in-memory data) while leave data resolution (i.e., behavior identification) to the analysis module, which runs in a separate PC.

Taking in and launching an app to be analyzed, NCScope instructs ETM to record the instructions executed by the app (see §3.1) and invoke eBPF programs to retrieve the data from the app's virtual memory and save it to a memory region resided in the kernel-space (see §3.2). Note that since the kernel configurations used for enabling eBPF (e.g., CONFIG_KPROBES, CONFIG_UPROBES) have already been turned on in the recently released Android systems (since Android 9.0), NCScope requires *no* modification to Android kernel. When the execution of native code ends, the data collection module outputs the instruction trace (e.g., ETM stream) and the memory data collected by eBPF to the analysis module for accomplishing various tasks, such as identifying self-protection and anti-analysis behaviors of native code (see §4.2 and §4.3), diagnosing memory corruption bugs (see §4.4), and measuring performance (see §5.5).

Assumption. We assume that the apps to be analyzed *cannot* gain the root privilege. That is, the apps can neither disable eBPF nor load a kernel module that inspects the values of the registers used for configuring ETM to detect the presence of NCScope. This assumption is rational because Android has already adopted various techniques to protect the integrity of its kernel [41], and smartphone vendors have also employed many approaches [48] to secure the kernels of their customized Android systems.

In this section, we describe the data collection module, including how NCScope traces the executed instructions of an app (see §3.1) and retrieves the accessed data from its virtual memory (see §3.2), because such run-time data can be leveraged to characterize its behaviors[63]. The analysis module will be described in §4.

3.1 Tracing Instructions

NCScope uses ETM to trace an app's executed instructions. By default, ETM records the target address of each indirect branch instruction executed by CPU. However, it is non-trivial to recover the app's behaviors from such an ETM stream due to two reasons. (1) The ETM stream contains a large amount of trace elements that are irrelevant to the analyzed app, because the target addresses of all indirect branch instructions executed by CPU will be recorded. (2) The ETM stream contains only the target address of each executed indirect branch instruction. To recover the function calls performed by the app, which is essential for recovering its behaviors, the target address of each executed direct branch instruction is also needed, because the execution of such an instruction may refer to a method invocation. To address these issues, NCScope enables the context ID tracing and branch broadcasting tracing of ETM. Table 1 lists the ETM registers used for enabling and configuring the tracing. We elaborate how NCScope adjusts these ETM registers as follows.

Table 1: A summary of registers used for configuring ETM.

ETM Registers	Field	Description
TRCACVR0/TRCACVR1 (Address Comparator Value Registers)	ADDRESS	Set the virtual memory range for branch broadcast tracing
TRCBBCTLR (Branch Broadcast Control Register)	RANGE	Set the address range comparator pair for branch broadcast tracing
TRCCIDCCTLR0 (Context ID Comparator Control Register)	COMP0	Enforce Context ID comparison with relevant byte in TRCCIDCVR0
TRCCIDCVR0 (Context ID Comparator Value Register)	VALUE	Set PID to be compared with during Context ID tracing
TRCCONFIGR (Trace Configuration Register)	CID	Validate context ID tracing
	BB	Validate broadcast tracing
TRCIDR0 (ID Register)	TRCBB	Examine whether branch broadcast tracing is supported

Note that, although the ARM platform provides an on-chip buffer for storing ETM stream, its capacity is insufficient (64KB at most [61]) to store the stream. Therefore, NCScope uses DSTREAM [6], a dedicated off-chip device with a 4GB buffer, to store ETM stream.

• **Context ID Tracing.** NCScope enables the context ID tracing so that ETM just traces the instructions executed by the process of app under analysis. Specifically, it sets the CID field of TRCCONFIGR to 0x1, assigns the target app's PID to the TRCCIDCVR0, and adjusts the COMP0 field of TRCCIDCCTLR0 to 0x1.

• **Branch Broadcast Tracing.** To record the target address of each executed direct branch instruction, NCScope enables branch broadcast tracing of ETM by setting the BB field of TRCCONFIGR to 0x1. To specify the memory range for branch broadcast tracing, NCScope assigns 0x0 and 0x7fffffffff to the ADDRESS field of TRCACVR0/1, and sets the RANGE field of TRCBBCTLR to 0x1.

3.2 Collecting Memory Data

The instruction trace alone is insufficient for native code analysis. For example, without the value referenced by the first parameter of the function open defined in the system library libc.so, we cannot know the file accessed by the app. Hence, NCScope leverages eBPF to collect the app's memory data at run-time, especially the parameter values of system functions called by the app. As shown in Fig. 1, NCScope is equipped with several eBPF programs to instruct Android kernel to collect the app's in-memory data. In the following, we introduce how NCScope retrieves the memory data.

• **Retrieving Memory Data.** NCScope instructs eBPF with the eBPF programs to hook multiple system functions for getting their parameter values at run-time. Since parameters can use pointers to access the data in memory, NCScope takes two steps to retrieve the referenced values. First, to fetch the data, NCScope obtains the base

address that stores the data in memory. Specifically, NCScope uses PT_REGS_PARM*, a series of macros provided by the kernel [33], to get the parameter value, which is the base address of the memory data. Second, since the data is stored at the base address obtained in the first step, NCScope fetches the data from the base address. Specifically, NCScope sends the base address to bpf_probe_read, an interface provided by bcc for instructing eBPF to read the bytes starting from the based address. For instance, to obtain the path of file accessed by calling open, in the first step, NCScope uses PT_REGS_PARM1 to get open's first parameter value, which is the base address of the file path string. Then, in the second step, relying on the base address, NCScope uses bpf_probe_read to retrieve the in-memory file path string.

• **Storing Memory Data.** The parameter values of system functions retrieved by eBPF will be saved in the kernel-space virtual memory [11]. In particular, we address the following two problems related to storing the data collected by eBPF.

First, since system functions are usually defined in shared libraries (e.g., libc.so), they will be invoked by different apps. Therefore, the retrieved memory data may contain the parameter values of system functions called by other apps instead of the app under analysis. Since the memory data accessed by other apps is irrelevant for analyzing the native code of the app under analysis and will make NCScope wrongly treat behaviors of other apps as behaviors of the app under analysis, we need to filter out them. To achieve this purpose, NCScope marks the source (i.e., PID) of the data retrieved by eBPF, so that NCScope can find the memory data related to the app under analysis according to the app's PID in the analysis module. Specifically, NCScope lets eBPF programs record the PID value when getting the parameter values of system functions by using bpf_get_current_pid_tgid, an interface provided by bcc, and saves it along with the retrieved data to the kernel-space memory.

Second, since the memory data accessed by the app forms a sequence according to the execution order of the system functions called by the app, and such sequence provides useful information for identifying specific behaviors (see §4.2 and §4.3), we add timestamps to the data retrieved by eBPF and then order the obtained memory data according to their timestamps. In detail, our eBPF programs call the bcc interface bpf_ktime_get_ns to get the time when they are run by eBPF to retrieve the system functions' parameter values, and saves it along with the retrieved data to the kernel-space memory.

4 ANALYSIS MODULE OF NCSCOPE

To bridge the semantic gap between the raw data (i.e., the collected ETM stream and memory data) and the high-level semantic information (e.g., function calls and data flow) required for native code analysis [54], NCScope recovers run-time function calls (see §4.1) and data flow (see §4.4) in the app's native code from the raw data. In §4.2-§4.4, we introduce how NCScope detects self-protection and anti-analysis mechanisms implemented in native code, and diagnoses use-after-free and double-free bugs in native code. §4.5 introduces how to extend NCScope with new analysis functionality.

4.1 Recovering Run-time Function Calls

NCScope resolves the ETM stream to find the system functions (including Android framework APIs and system library functions) that have been called by the app at run-time because they provide the necessary semantic information for native code analysis. Since existing tools (e.g., ptm2human [7], ds-5 [5]) cannot accomplish this task, we develop a new ETM stream resolver for NCScope. It first constructs the mapping from each instruction address to the corresponding system function for determining the system functions called by the app from the traced instruction stream. Then, from the constructed mapping, it finds the system functions called by the app according to the *Address* elements stored in the ETM stream. More details are presented as follows.

• **Constructing Instruction-Function Mapping.** To determine the system functions called by the app from the instruction addresses tracked by ETM, NCScope first constructs a mapping between system functions and the addresses of their instructions by analyzing the memory map of the app and the disassembled information of system libraries and framework OAT files.

In detail, NCScope gets the memory map information about the loaded system libraries and framework OAT files from the memory map (i.e., /proc/pid/maps) of the app. From the obtained memory map information, NCScope finds the start address V_m of each executable memory region for calculating the address of each instruction mapped in the virtual memory. Then, NCScope uses objdump [31] and oatdump [30] to disassemble system libraries and framework OAT files, respectively, in order to extract the file offset F_i of each instruction. Since an instruction's file offset F_i and its virtual offset V_i can be different, we further calculate the difference δ between the two offsets through Equation (1). Specifically, for instructions in the system library, NCScope uses objdump to get the virtual memory offset V_t and the file offset F_t of the library's .text section. For instructions in the framework OAT file, NCScope uses oatdump to get the file offset F_o of the OAT file's .oatexec section. After δ is calculated, NCScope uses Equation (2) to get the address V_i of each instruction mapped in the virtual memory.

$$\delta = \begin{cases} F_t - V_t & (\ system\ library\ ) \\ 0 - F_o & (\ framework\ OAT\ file\ ) \end{cases} \quad (1)$$

$$V_i = V_m + (\ F_i + \delta\ ) \quad (2)$$

For example, to calculate the virtual address (V_i) of the first instruction of the system function __android_log_print defined in the system library liblog.so to build the mapping between V_i and __android_log_print. In particular, from the memory map file of the app, we find that the executable section (i.e., .text section) of liblog.so is mapped in the virtual memory 0x700148f000 (V_m). Then, we use objdump to get the disassembled information about liblog.so, including the file offset of the first instruction of __android_log_print 0x7c38 (F_i), the virtual memory offset of .text section of liblog.so 0x4758 (V_t), and the file offset of .text section of liblog.so 0x4758 (F_t). According to Equations (1) and (2), we calculate $V_i = V_m + (F_i + F_t - V_t)$ = 0x7001496c38 and get an entry of the instruction-function mapping 0x7001496c38 -> __android_log_print.

• **Finding System Functions Called.** Since the ETM stream records the virtual addresses of instructions executed by the app at run-time, relying on the instruction-function mapping, NCScope can recover the app's run-time function calls by mapping instruction

addresses recorded in the ETM stream to system functions. Specifically, NCScope gets the virtual address of the tracked instruction from each Address element recorded in the ETM stream. Since the Address element uses its right-most bit to record the instruction set of the tracked instruction [16], NCScope parses the remaining bits to get the virtual address of the instruction. Then, NCScope queries the instruction-function mapping to check whether the recorded instruction address is associated with a system function. For example, to decide whether the app has called `__android_log_print` at run-time, we first check if the instruction-function mapping has the entry 0x7f12345678 ↦ `__android_log_print`, because we rely on this mapping relationship to determine whether the function is called. If that is the case, 0x7f12345678 is the virtual address of an instruction in `__android_log_print`. Then, NCScope resolves the addresses of instructions tracked by ETM. If we find that the address of an instruction tracked by ETM is 0x7f12345678, according to the entry of the instruction-function mapping, we can determine that `__android_log_print` is called by the app at run-time.

• **Determining System Function Called by Native Code.** Since system functions can be invoked by either Java code or native code of the app, NCScope further determines the functions called by native code. Since an app can have two types of native functions, i.e., `JNI_OnLoad` called by the system when loading native libraries and other common native functions called by the app's JNI functions [51], we introduce how NCScope determines the system functions called by these two types of native functions as follows.

▹ **JNI_OnLoad.** Since `JNI_OnLoad` is called by the system function `JavaVMExt::LoadNativeLibrary` defined in the system library `libart.so`, NCScope leverages this observation to find the system functions executed by `JNI_OnLoad`. Specifically, NCScope uses the start address L_s of `LoadNativeLibrary` and the address L_e of the instruction in `LoadNativeLibrary`, to which `JNI_OnLoad` returns, to divide the instruction trace. The system functions appear between a pair of L_s and L_e are those called by `JNI_OnLoad`.

To find the parameter values of system functions called by `JNI_OnLoad` from the data retrieved by eBPF, NCScope uses eBPF programs to hook L_s and L_e to get their timestamps T_s and T_e, which indicate when these two instructions are executed. Relying on the timestamps, NCScope gets the time slots for executing `JNI_OnLoad`. The memory data obtained within the time slots are the parameter values of system functions called by `JNI_OnLoad`.

▹ **Common Native Functions.** The app's common native functions, excluding `JNI_OnLoad`, can only be accessed by the app's JNI functions. Since Android framework invokes the system function `art::artQuickGenericJniTrampoline` defined in the system library `libart.so` before the execution of each JNI function and then invokes the function `art::artQuickGenericJniEndTrampoline` after the execution of the JNI function, NCScope leverages this observation to find the system functions executed by common native functions. More specifically, NCScope uses the start address J_s of `artQuickGenericJniTrampoline` and the start address J_e of `artQuickGenericJniEndTrampoline` to divide the instruction trace. The system functions found between J_s and J_e are those called by the app's common native functions.

Similarly, to find the parameter values of system functions called by the app's common native functions, NCScope uses eBPF programs to hook J_s and J_e to get the time slots for executing the app's common native functions. The in-memory data obtained within the time slots are the parameter values of the system functions called by the app's common native functions.

4.2 Detecting Self-Protection Methods

We investigate self-protection methods that can be implemented in native code by studying the code snippets presented in the guidelines [32] and Github repositories [34, 35, 38]. Eventually, we collect 10 self-protection methods in 2 types, including root detection and tampering detection. For each method, we design a detection rule for NCScope to identify it. Table 2 summarizes these rules, where the symbol **FUNC**(f) indicates that the system function f is found in the recovered system functions called by the app's native code, and **ARGx**(f) denotes that NCScope uses eBPF to record the value of function f's xth parameter. It is worth noting that although we try our best to collect as many self-protection methods as possible, we do not claim that the rules in Table 2 are comprehensive. Instead, we use them to demonstrate the usage of NCScope in identifying commonly-used self-protection methods implemented in native code. We describe how to add new rules for discovering new self-protection methods in §4.5.

• **Root Detection (RTD).** Since attackers can gain control of apps running in rooted devices, security-sensitive apps employ the following RTD methods to avoid running in rooted devices [59, 60, 70].

▹ **RTD-1:** Apps look for the executable files (e.g. `/system/xbin/su`) or APKs (e.g., `/system/app/Superuser.apk`) used for gaining or managing the root privilege from the file system, because these files are typically found on rooted devices [32]. However, if names of these files are obfuscated, RTD-1 will fail.

▹ **RTD-2:** Apps check whether the shell command `su` can be executed, because this command is typically existed in rooted systems [32]. However, if the name of the command is changed, RTD-2 will fail.

▹ **RTD-3:** Apps examine whether the apps used for rooting the device (e.g., KingRoot [26]) or managing the root-privilege (e.g., Superuser [39]) have been installed on the device, because these apps are typically installed on rooted devices [32]. However, if names of these apps are obfuscated, RTD-3 will fail.

▹ **RTD-4:** Apps execute the command `mount` to inspect whether the permissions of mounted system directories (e.g., `/system/xbin`) are writable, because writable permissions on system directories usually indicate rooted devices [32].

▹ **RTD-5:** Apps check the system's build tag [32], because it indicates whether the system is a custom ROM that is commonly rooted [15].

▹ **RTD-6:** Apps examine special system properties (e.g., `ro.secure`), because they indicate whether the systems are rooted [32].

▹ **RTD-7:** Apps execute the shell command `ps` to find whether daemon services of rooting apps (e.g., `daemonsu` [40]) are running, because these services are typically running on rooted devices [32]. However, if names of these services are obfuscated, RTD-7 will fail.

Detection methods. **RTD-1:** Since the `libc` functions, including `access`, `open`, or `stat`, can be invoked to check files' existence, NCScope records the values of their first parameters, which store the file paths, and then inspects the obtained paths to find whether the app searched for rooting related executable files or APKs.

RTD-2: Native code can call either `execvp` or `execvpe` exported by `libc.so` to execute the shell command. Thus, NCScope records the

Table 2: The rules used by NCScope to identify self-protection behaviors.

Type	Id	Rules	Notes
RTD	1	(1) **ARG1**(`libc.open`) == "/system/xbin/su" (or paths of rooting apps "/system/app/Superuser.apk")	or **ARG1**(`libc.access`), **ARG1**(`libc.stat`)
	2	(1) **ARG1**(`libc.execvp`) == "su" (or the paths of the `su` file)	or **ARG1**(`libc.execvpe`)
	3	(1) **ARG1**(`libc.strstr`) == "superuser" (or other names of rooting apps)	or **ARG2**(`libc.strstr`)
	4	(1) **ARG1**(`libc.execvp`) == "mount" (2) **ARG1**(`libc.strstr`) == "/system/xbin" (or other system directories)	or **ARG1**(`libc.execvpe`) or **ARG2**(`libc.strstr`)
	5	(1) **ARG2**(`libart.FindClass`) == "android/os/Build" (2) **ARG3**(`libart.GetStaticFieldID`) == "TAGS" (3) **ARG1**(`libc.strstr`) == "test-keys"	 or **ARG2**(`libc.strstr`)
	6	(1) **ARG1**(`libc.__system_property_get`) == "ro.debuggable" (or "ro.secure")	
	7	(1) **ARG1**(`libc.execvp`) == "ps" (2) **ARG1**(`libc.strstr`) == "daemonsu" (or the paths of rooting apps)	or **ARG1**(`libc.execvpe`) or **ARG2**(`libc.strstr`)
TPD	1	(1) **ARG2**(`libart.FindClass`) == "android/content/pm/PackageInfo" (2) **ARG3**(`libart.GetFieldID`) == "signatures"	
	2	(1) **ARG2**(`libart.NewStringUTF`) == "classes.dex" (2) **FUNC**(`java.util.zip.ZipFile.<init>`) (3) **FUNC**(`java.util.zip.ZipEntry.getCrc`)	or **ARG2**(`libart.NewString`)
	3	(1) **FUNC**(`android.content.pm.PackageManager.getInstallerPackageName`) (2) **ARG1**(`libc.strstr`) == "com.android.vending" (or package names of other app installers)	 or **ARG2**(`libc.strstr`)

[1] Note that, since `strcmp`, `strcasecmp`, `strncmp`, `strncasecmp` and `strstr` have quite similar functionalities, any one of them can be replaced by the others.

first parameter values of these functions, which store the executed shell commands, to see whether `su` was executed by the app.

RTD-3: Since the `libc` functions, including `strcmp`, `strcasecmp`, `strncmp`, `strncasecmp`, or `strstr`, can be called to conduct the string comparison, the app can use any of them to find rooting related apps in the string list that stores the paths of the installed apps. Hence, `NCScope` records the parameter values of these functions.

RTD-4: `NCScope` analyzes parameters of functions for shell command execution and string comparison to see whether the app retrieved and examined permissions of mounted system directories.

RTD-5: To access the field of a Java class or a Java object in native code, the app will call the `libart` functions `art::JNI::FindClass` and `art::FindFieldID`. Based on this observation, `NCScope` obtains the values of the second parameter of `FindClass` and the third parameter of `FindFieldID`, which store the names of the Java class and field, to see whether the app accessed the `TAGS` field of the Java class `android.os.Build` to get the build tag of system.

RTD-6: The function `__system_property_get` defined in `libc.so` can be called to access and get the values of system properties. Accordingly, `NCScope` records the value of this function's first parameter, which provides the name of the system property. Then, from the obtained values, `NCScope` finds whether the system properties, the values of which disclose that the system is a custom one, were accessed by the app.

RTD-7: `NCScope` inspects parameters of the functions for shell command execution and string comparison to see whether the app examined the names of running processes to find the daemon services of rooting apps.

• **Tampering Detection (TPD).** Since repackaging is a major threat to the app ecosystem [73, 84], apps usually employ the following `TPD` methods to protect themselves from being tampered.

▹ **TPD-1:** Apps obtain their signatures to detect repackaging, because original apps and repackaged apps have different signatures [32].

▹ **TPD-2:** Apps perform the cyclic redundancy check (CRC) over their Dex files to check whether they have been tampered, because original apps and tampered apps have different CRC [32].

▹ **TPD-3:** Apps examine their installers [58] to prevent them from being installed from suspicious sources (e.g., third-party markets), because suspicious sources are flooded with repackaged apps [100].

Detection methods. TPD-1: Since the field `signatures` of the app's `PackageInfo` object stores the app's signature, `NCScope` checks the parameter values of `FindClass` and `FindFieldID` to see whether this field was accessed by the app.

TPD-2: The app can use JNI reflection to invoke the framework APIs `ZipFile.<init>` and `ZipEntry.getCrc` to calculate the CRC of its Dex files. Accordingly, `NCScope` analyzes the functions called by the app's native code to see whether these APIs were invoked. Since the names of Dex files are passed to the second parameter of the functions `NewStringUTF` or `NewString` defined in `libart.so` during the calculation, `NCScope` inspects parameters of these functions to ensure that CRC was conducted over the app's Dex files.

TPD-3: Since the method `getInstallerPackageName` of the class `PackageManager` can be used to get the package name of the app's installer, `NCScope` analyzes the system function calls to see whether the API was invoked by the app. Then, `NCScope` retrieves and inspects the parameter values of the functions for string comparison to find whether the package names of legal app markets (e.g., "com.android.vending" for Google Play store) are compared with the package name of the app's installer.

4.3 Detecting Anti-Analysis Mechanisms

We examine anti-analysis mechanisms that can be implemented in native code by referring to the code segments and technical details in the research papers [62, 72, 78, 94], whitepapers [29, 32], and `Github` repositories [3, 4] and collect 12 anti-analysis mechanisms in 5 types, including debugger detection, emulator detection, DBI framework detection, timing check, and dynamic code loading. For each mechanism, we design a detection rule for `NCScope` to identify it. Table 3 lists these rules. It is worth noting that although we try our best to collect as many anti-analysis mechanisms as possible, we do not claim that the rules in Table 3 are complete. Instead, we use them to demonstrate the ability of `NCScope` in identifying

widely-used anti-analysis mechanisms implemented in native code. In the following, we present these anti-analysis mechanisms and how NCScope detects them. We describe how to add new rules for discovering new anti-analysis mechanisms in §4.5.

• **Debugger Detection (DBD).** Apps usually use the following DBD mechanisms to detect the presence of debuggers to prevent it from being analyzed by them [32, 62, 72, 94].

▹ **DBD-1:** Apps leverage JNI reflection to call the framework API `Debug.isDebuggerConnected`, because it is commonly used to detect JDWP-based debuggers [32, 94].

▹ **DBD-2:** Apps inspect the content of its process status files (i.e., `/proc/pid/status` or `/proc/self/status`), because the content indicates whether `ptrace` based debuggers have attached [62, 72].

▹ **DBD-3:** Apps check the system property debug.atrace.tags.enableflags, because it indicates the presence of `ftrace` [46].

Detection methods. DBD-1: NCScope analyzes the system functions called by the app's native code to determine whether the function `Debug.isDebuggerConnected` was invoked.

DBD-2: From the parameter values of `open` and the functions for string comparison, NCScope finds out whether the app opened its process status file and searched "TracerPid:" in the file content to detect the presence of `ptrace` based debuggers.

DBD-3: NCScope examines parameters of `__system_property_get` to see whether the `ftrace` associated system property was accessed.

• **Emulator Detection (EMD).** Since many dynamic analysis tools [87, 93, 98] are built upon Android emulator (e.g., QEMU), apps usually adopt the following EMD mechanisms to detect the presence of emulators to avoid being analyzed in such environments [32, 62, 94].

▹ **EMD-1:** Apps check the content of special system files (e.g., /proc/tty/drivers), because it is specially introduced by emulators [62, 94].

▹ **EMD-2:** Apps inspect the existence of system properties (e.g., `init.svc.qemud`) that are introduced by emulators [94].

▹ **EMD-3:** Apps call the APIs (e.g., `getSubscriberId`) defined in `TelephonyManager` and check their return values (e.g., 310260000000000), because these values are specially used in emulators [32].

Detection methods. EMD-1: NCScope analyzes the parameter values of `open` and the functions for string comparison to see whether the app accessed and inspected the content of particular system files to find the data introduced by emulators.

EMD-2: NCScope examines parameters of `__system_property_get` to discover whether the app checked the presence of special system properties introduced by emulators.

EMD-3: NCScope checks the recovered system function calls to see whether the `TelephonyManager` APIs were invoked. Then, it inspects the parameter values of the functions for string comparison to examine whether the return values of these APIs were compared with those from emulators.

• **DBI Framework Detection (DFD).** To prevent being instrumented, apps usually adopt the following DFD mechanisms to detect DBI frameworks [32, 94].

▹ **DFD-1:** Since DBI frameworks load several artifacts (e.g., `.so` files) to memory, apps examine the memory map (i.e., `/proc/pid/maps`) to look for such artifacts [32, 94].

▹ **DFD-2:** Since DBI frameworks will open specific TCP ports (e.g., TCP port 27042 is opened and used by `frida` [19]) to communicate with their controllers running on other devices, apps can execute the shell command `netstat` or `ss` to check such opened ports [32].

Detection methods. DFD-1: NCScope analyzes the parameter values of `open` to see whether the app accessed its memory map file. Then, NCScope dissects the parameter values of the functions for string comparison to determine whether the app searched suspicious strings (e.g., "frida", "xpose") to find the memory mapped artifacts.

DFD-2: NCScope inspects the parameter values of the functions for shell command execution and string comparison to see whether the app retrieved and examined the network states to find the TCP ports opened by DBI frameworks.

• **Timing Check (TCK).** Since dynamic analysis tools slowdown the execution of apps [29, 32], apps can compute the time spent on executing a special task [88] to infer the presence of dynamic analysis tools. Commonly, apps can call the function `time` or `gettimeofday` defined in `libc.so` to get the time value (i.e., **TCK-1**).

Detection method. TCK-1: To calculate the time consumption of a task, `gettimeofday` or `time` will be called at least twice, one for getting the task's start time and the other for getting the task's end time. Based on this observation, NCScope examines the system functions called by the app's native code to find whether the system functions for getting the time were continuously called.

• **Dynamic Code Loading (DCL).** To prevent the code from being reverse engineered by static analysis tools [21, 44, 56, 83], apps usually hide their critical code in APKs and dynamically load them to memory before execution. Commonly, there are three approaches for apps to load the bytecode or native code at run-time [62, 78, 94].

▹ **DCL-1:** Apps leverage JNI reflection to call the framework APIs (e.g., `DexClassLoader.<init>`, `DexPathList.makePathElements`, and `DexFile.loadDex`) for loading extra Dex files dynamically [78, 94].

▹ **DCL-2:** Apps call `mmap` defined in `libc.so` to apply for writable and executable memory regions by setting the third parameter of `mmap` to 0x5 [42, 62] to store the loaded native code.

▹ **DCL-3:** Apps call `mprotect` defined in `libc.so` to give the writable property to executable but non-writable memory regions by setting the third parameter of `mprotect` to 0x4 [53, 62], so that native code can be loaded into such memory regions.

Detection methods. DCL-1: NCScope inspects recovered function calls to check whether the APIs for loading Dex files were called.

DCL-2: NCScope gets the third parameter value of `mmap`, which specifies the property of the applied memory region. From the retrieved values, NCScope determines whether the app applied for writable and executable memory regions to store the released native code.

DCL-3: NCScope obtains the third parameter value of `mprotect`, which indicates the property given to the memory region. Then, NCScope inspects the obtained values to find whether the app gave the writable property to memory regions.

4.4 Diagnosing Memory Corruption Bugs

NCScope can diagnose two types of memory corruption bugs in the app's native code, including CWE-416 (use-after-free) [50] and CWE-415 (double-free) [49], by recovering its run-time data flow. In particular, with the run-time execution trace (i.e., ETM stream), NCScope employs offline symbolic execution [75, 96] to recover the app's run-time data flow. Note that, we implement this module by adapting the existing work [96], targeting at x86 platform, to ARM.

• **Offline Symbolic Execution.** Taking in the initial program state (including registers, stack, and heaps) and the execution trace

Table 3: The rules used by NCScope to identify anti-analysis behaviors.

Type	Id	Rules	Notes
DBD	1	(1) **FUNC**(`android.os.Debug.isDebuggerConnected`)	
	2	(1) **ARG1**(`libc.open`) == "/proc/pid/status" (or "/proc/self/status") (2) **ARG1**(`libc.strstr`) == "TracerPid:"	or **ARG2**(`libc.strstr`)
	3	(1) **ARG1**(`libc.__system_property_get`) == "debug.atrace.tags.enableflags"	
EMD	1	(1) **ARG1**(libc.open) == "/proc/tty/driver" (2) **ARG1**(`libc.strncmp`) == "goldfish"	or **ARG2**(`libc.strncmp`)
	2	(1) **ARG1**(`libc.__system_property_get`) == "init.svc.qemud" (or other system properties introduced by `QEMU`)	
	3	(1) **FUNC**(`android.telephony.TelephonyManager.getSubscriberId`) (or other `TelephonyManager` APIs) (2) **ARG1**(`libc.strcmp`) == "310260000000000" (or other suspicious values)	or **ARG2**(`libc.strcmp`)
DFD	1	(1) **ARG1**(libc.open) == "/proc/pid/map" (or "/proc/self/maps") (2) **ARG1**(libc.strstr) == "frida" (or other typical strings in the names of files related to DBI frameworks)	or **ARG2**(libc.strstr)
	2	(1) **ARG1**(`libc.execvp`) == "netstat" (or "ss") (2) **ARG1**(libc.strcmp) == "27042" (or other TCP port numbers used by DBI frameworks)	or **ARG1**(`libc.execvpe`) or **ARG2**(libc.strcmp)
TCK	1	(1) **FUNC**(libc.gettimeofday) (see §4.3 for more details)	or **FUNC**(libc.time)
DCL	1	(1) **FUNC**(`dalvik.system.DexClassLoader.<init>`) (or other functions for loading the Dex files `DexFile.loadDex`)	
	2	(1) proc**ARG3**(`libc.mmap`) & 0x5 == 0x5	
	3	(1) **ARG3**(`libc.mprotect`) & 0x4 == 0x4	

[1] Note that, since `strcmp`, `strcasecmp`, `strncmp`, `strncasecmp` and `strstr` have quite similar functionalities, any one of them can be replaced by the others.

Table 4: The measurement on the additional overhead incurred by NCScope.

	Integer Score			Floating Point Score			Memory Score			Overall Score			Start-up Time		
	Mean	STD	Slowdown	Mean	STD	Slowdown	Mean	STD	Slowdown	Mean	STD	Slowdown	Mean	STD	Slowdown
Baseline	1114.6	3.6	1.000x	777.0	7.5	1.000x	1606.4	11.1	1.000x	1087.3	4.2	1.000x	2.50	0.06	1.000x
NProfiler$_{ins}$	1100.2	8.7	1.013x	777.0	4.6	1.000x	1601.7	16.8	1.003x	1079.7	4.7	1.007x	2.51	0.06	1.004x
NProfiler$_{mem}$	766.8	3.5	1.455x	774.5	3.0	1.003x	1567.4	10.5	1.025x	925.4	1.3	1.175x	2.94	0.04	1.176x
NProfiler$_{ins+mem}$	765.9	4.8	1.455x	770.2	6.6	1.009x	1556.9	4.1	1.032x	921.2	2.5	1.180x	3.00	0.02	1.200x
DroidScope$_{ins}$	25.2	3.0	44.230x	16.4	2.3	47.378x	136.6	3.6	11.760x	43.4	2.8	25.053x	43.82	2.38	17.528x
DroidScope$_{ins+mem}$	12.5	1.5	89.168x	10.8	1.1	71.944x	83.9	5.8	19.147x	25.9	1.9	41.981x	118.39	5.08	47.356x

recorded at run-time, offline symbolic execution recovers the run-time data flow by tracking data values alongside each instruction in the execution trace. For the data (e.g., return values of system calls) that cannot be inferred from the initial program state, they will be symbolized and then propagated through the execution trace.

Specifically, since the run-time execution flow of the app's native code is recorded in the ETM stream, to apply the offline symbolic execution to recover the run-time data flow, `NCScope` leverages eBPF to retrieve the initial program state just before the execution of native code. In detail, since the execution flow of the app's native code begins with the start address of `LoadNativeLibrary` or `artQuickGenericJniTrampoline` (see §4.1), `NCScope` uses eBPF programs to hook these addresses to obtain the register values and the in-memory data of the app's stack and heaps. Note that, we observe that every app's stack will be stored in a fixed virtual memory region. Based on this observation, we get the address and size of stack via any running app's memory map and instruct eBPF to retrieve the data in stack. Meanwhile, since the app invokes allocation functions (e.g., `malloc` defined in `libc.so`) [27] or the system call `mmap` [42] to apply for heaps, `NCScope` uses eBPF programs to hook these functions to get the addresses and sizes of allocated heaps, and then instructs eBPF to retrieve the data in heaps.

• **Use-after-free and Double-free Detection.** During the offline symbolic execution, `NCScope` monitors all calls to memory allocation functions and free functions (e.g., `free` defined in `libc.so`) [27], and saves the allocated and freed buffers to an allocation list and a free list, respectively. More precisely, when a memory allocation call is reached, the size argument P_{size} is extracted and the symbolized address pointer $P_{address}$ (referring to the address of the allocated buffers) is assigned with a concrete value, and the entry $<P_{address}, P_{size}>$ is saved to the allocation list. When a free call is reached, the address pointer parameter (referring to the address of the freed buffers) is extracted and the corresponding entry is moved from the allocation list to the free list. To detect use-after-free, for each instruction in the execution flows, addresses accessed by memory operations are checked against the free list. If existed, a use-after-free bug is found. To detect double-free, for each call to free functions, addresses of the freed buffers are checked against the free list. If existed, a double-free bug is found.

4.5 Extending NCScope

• **Adding new rules for detecting self-protection methods and anti-analysis mechanisms.** Analysts can perform three steps to add new rules for detecting new self-protection methods and/or anti-analysis mechanisms. First, according to the implementations of the new self-protection methods or anti-analysis mechanisms, analysts decide the necessary system functions and parameter values that can be used for detecting them. Second, if the necessary parameter values of the system functions have not been recorded (i.e., they are not included in Table 2), `NCScope` takes in the eBPF programs written by analysts or re-use our developed eBPF programs to instruct eBPF to record such memory data. Third, to detect

the new self-protection methods or anti-analysis mechanisms, from the function calls recovered by NCScope, analysts can find whether the necessary system functions are called. Meanwhile, from the memory data recorded by NCScope, analysts inspect whether the necessary parameter values of system functions are accessed.

• **Diagnosing new memory corruption bugs.** Analysts can extend this module to diagnose new memory corruption bugs by introducing new detection rules (e.g., those introduced in the existing work [96]) when performing offline symbolic execution.

• **Adding new functionality.** Analysts can add new functionality to NCScope by extending the analysis module to inspect system functions called and memory data accessed by native code of apps at run-time. Specifically, since NCScope can recover all the system functions called by native code of apps (see §4.1), analysts just need to customize eBPF programs, which are then taken by NCScope to instruct eBPF to retrieve the needed memory data. Since bcc [10] provides convenient APIs for writing eBPF programs, our experiences show that it is non-difficult to extend NCScope with new functionality. Our source code can be used as a reference.

5 EVALUATION

We implement the prototype of NCScope with about 7k lines of Java code and 3k lines of C/Python code based on: ds-5 [5], a tool for retrieving the ETM stream from ARM platform; adeb [1], a tool that provides interfaces for accessing the functionalities of eBPF; and ARCUS [96], a tool for conducting offline symbolic execution. NCScope is deployed on a Juno r2 development board [24], running an Android 9.0 system with the 4.14.59 Android common kernel. We evaluate the performance and functionalities of NCScope by answering the following five research questions (RQs).

RQ1: How is the overhead incurred by NCScope?
RQ2: How prevalent are self-protection behaviors implemented in native code of financial Android apps?
RQ3: How prevalent are anti-analysis behaviors implemented in native code of Android malware?
RQ4: Can NCScope assist the diagnose of memory corruption bugs in the app's native code?
RQ5: Can NCScope assist the analysis on the performance of the app's native code?

5.1 RQ1: Overhead

The extra overhead introduced by NCScope comes from two aspects: tracing the executed instructions (using ETM); and retrieving the in-memory data (leveraging eBPF). In Table 4, we use the symbols *ins* and *mem* to denote the two operations, respectively. For each operation, we run the native code tests provided by the benchmark app Geekbench [20] 10 times to measure the performance impact of the operation. Specifically, we calculate the average of integer scores, floating point scores, memory scores, and overall scores assessed by Geekbench, and record the benchmark app's average start-up time in seconds. We also calculate the performance scores when both of the operations and none of the operations are conducted by NCScope, respectively, and the latter is treated as the baseline. Note that, a higher score denotes a better performance.

Result: Table 4 shows the results, where the Slowdown column provides the times of slowdown brought by NCScope. We observe that ETM (i.e., instruction tracing) will not slowdown the execution of app under analysis, and eBPF (i.e., memory data retrieving) only brings 1.175x slowdown to the overall score. When both the instruction tracing and memory data retrieving are conducted by NCScope, it just causes 1.180x slowdown to the overall score.

We further compare the performance of NCScope with DroidScope [98], a widely adopted emulator based dynamic app analysis platform that can perform the same operations (i.e., instruction tracing and memory data retrieving) as NCScope. The PC for running DroidScope is equipped with Intel i7-6700k CPU and 64GB RAM. Note that, since DroidScope conducts memory data retrieving together with instruction tracing, we do not separately measure the performance impact of memory data retrieving. According to the results, we see that either of instruction tracing and memory data retrieving of DroidScope bring obvious slowdown so that it cannot evade timing checks. When both the operations are conducted, DroidScope causes 41.981x slowdown to the overall score, indicating that DroidScope incurs much more slowdown than NCScope.

Answer to RQ1: NCScope introduces very small additional overhead to the execution of app under analysis.

5.2 RQ2: Prevalence of Self-Protection Methods

Data Set: We download more than 900 randomly selected financial apps from Google Play and pick out 500 samples that have native code, including 170 mobile banking apps, 152 digital wallet apps, 21 money transfer apps, 41 cryptocurrency apps, 31 personal loan apps, 20 insurance apps, and 65 stock trading apps. We apply NCScope to profile the self-protection behaviors implemented in the app's native code. It is worth noting that, we have manually run these financial apps on a rooted device and discovered that all of them cannot properly be launched and run on such the device, which implies that all of these apps will immediately execute self-protection methods (i.e., root detection) when they have been launched.

Table 5: A summary of identified self-protection behaviors.

Behavior	#app	Ratio	Behavior	#app	Ratio
RTD-1	77	15.4%	RTD-6	2	0.4%
RTD-2	105	21.0%	RTD-7	13	2.6%
RTD-3	2	0.4%	TPD-1	9	1.8%
RTD-4	56	11.2%	TPD-2	1	0.2%
RTD-5	0	0.0%	TPD-3	0	0.0%

Result: Table 5 shows the results, where #app denotes the number of financial apps that implement a specific self-protection behavior in their native code, and Ratio indicates the ratio of financial apps that implement a specific self-protection method in the evaluation.

Specifically, we discover that only 134 (26.8%) financial apps under analysis implement at least one self-protection method in their native code, which implies that the security of financial apps is far from expected. Moreover, 130 apps adopt root detection methods to prevent them from running in rooted devices, and RTD-1, RTD-2, and RTD-4 are the mostly adopted self-protection methods. However, only 10 (2%) financial apps under examination implement

tampering detection methods in native code, which suggests that the remaining apps are under the risk of repackaging attacks [69].

Precision & Recall: To assess the performance of NCScope in identifying the app's self-protection behaviors in native code, we download 20 randomly selected open-source apps from F-Droid [18], which implement neither self-protection nor anti-analysis methods. Applying NCScope to these apps, no self-protection behaviors are found, which indicates that there is no false positives. Moreover, we randomly select 20 financial apps that implement self-protection methods in their native code. By disassembling their native code, we manually analyze the obtained ARM instructions and find that each self-protection behavior is correctly detected by NCScope. That is, there is no false negatives.

Answer to RQ2: NCScope can precisely identify the app's self-protection behaviors in native code. It finds that only 26.8% of the evaluated financial apps implement self-protection methods in native code, implying that their security is far from expected.

5.3 RQ3: Prevalence of Anti-Analysis Methods

Data Set: We obtain 450 malicious apps from a security company and find that 300 of them have native code and can properly run on NCScope. We apply NCScope to profile their anti-analysis behaviors implemented in native code.

Table 6: A summary of identified anti-analysis behaviors.

Behavior	#app	Ratio	Behavior	#app	Ratio
DBD-1	2	0.7%	DFD-1	1	0.3%
DBD-2	225	75.0%	DFD-2	0	0.0%
DBD-3	0	0.0%	TCK-1	226	75.3%
EMD-1	1	0.3%	DCL-1	234	78.0%
EMD-2	1	0.3%	DCL-2	15	5.0%
EMD-3	1	0.3%	DCL-3	21	7.0%

Result: Table 6 presents the detailed results on profiling the anti-analysis behaviors of the malicious apps under analysis.

Specifically, NCScope identifies 235 (78.3%) malicious apps that implement anti-analysis methods in their native code, implying that Android malware commonly have anti-analysis behaviors. Meanwhile, we discover that DBD-2, TCK-1, and DCL-1 are the top-three adopted anti-analysis methods. TCK-1 can impede the analysis of existing dynamic app debugging, monitoring, or profiling tools (e.g., DroidScope) as they cause significant slowdown. NCScope can evade TCK-1 because it introduces very small additional overhead. Moreover, DCL-1 can hinder static analysis approaches as they do not take the dynamically released code into analysis. However, NCScope can help analysts capture the code released at run-time. Therefore, NCScope is very useful to malware analysis. We manually inspect the native code of malware with anti-analysis behaviors and find that they are heavily obfuscated, making it extremely hard for manual inspection. Hence, we use the apps from F-Droid and the apps developed by us to further evaluate NCScope as follows.

Precision & Recall: To assess the precision and the recall in detecting anti-analysis behaviors in native code, we reuse the 20 F-Droid apps in §5.2. Applying NCScope to these apps, no anti-analysis behaviors are detected, suggesting that there is no false positives. Moreover, since it is non-trivial and time-consuming to manually analyze the native code of malware due to heavy code obfuscation, we first randomly select 10 malware samples that implement anti-analysis methods in their native code. Then, after disassembling their native code, we manually inspect the obtained ARM instructions and find that each anti-analysis behavior is successfully detected by NCScope. That is, there is no false negatives.

Answer to RQ3: NCScope can precisely identify anti-analysis behaviors in native code. It finds that at least 78.3% of the evaluated malware implements anti-analysis methods in native code.

5.4 RQ4: Memory Corruption Diagnosis

Data Set: From NIST C/C++ Juliet suite [23], a collection of open-source test cases for CWEs, we adapt 20 buggy and 40 bug-free test cases for CWE-416 (use-after-free) and CWE-415 (double-free) to 60 Android apps' native code. Then, we apply NCScope to diagnose memory corruption bugs in these apps' native code.

Result: NCScope identifies all use-after-free and double-free bugs in native code of the apps under evaluation with no false positives and false negatives, indicating that it can precisely diagnose these memory corruption bugs in the app's native code. It is worth noting that, since use-after-free and double-free bugs may not crash the app [22], no exceptions will be raised by Android system so that these bugs cannot be diagnosed by analyzing the exceptions.

To demonstrate NCScope's capability of diagnosing memory corruption bugs in real-world apps, we apply NCScope to an F-Droid app [9], whose library `libpl_droidsonroids_gif.so` is vulnerable to CVE-2019-11932 [12]. NCScope correctly finds out that the vulnerability is caused by double-free.

Answer to RQ4: NCScope can aid memory corruption diagnosis by identifying use-after-free and double-free in native code.

5.5 RQ5: Performance Analysis

We conduct a case study to show the usefulness of NCScope to performance analysis of native code. NCScope finds out that to write a large amount of data to a file using the system function `fwrite` costs less time than using `write` [14], both of which are defined in `libc.so`. To compare the performance of the two functions, we develop an app with two JNI functions F_{fwrite} and F_{write}, which call `fwrite` and `write`, respectively. In each JNI function, the app opens an empty file, writes n (256) bytes data to the file for t (1,000) times, and then closes the file. As introduced in §4.1, since NCScope records the timestamps when the app starts to execute a JNI function (T_s) and exits from the same JNI function (T_e), we compute the execution time of a JNI function via the formula T_e - T_s. We run the app 10 times and discover that the average execution time of F_{fwrite} is 2.63 milliseconds, while the average execution time of F_{write} is 16.72 milliseconds. From the results, we notice that `fwrite` has a better performance than `write` in this case.

Analysis: With the retrieved run-time information, NCScope can aid analysts to find out the root cause of the performance difference. Specifically, from the recovered functions called by F_{fwrite}, we find that `fwrite` internally calls `write`, and F_{fwrite} calls `write` 63 times in total. Meanwhile, from the retrieved parameter values of `write`, we discover that, each of the prior 62 function calls to `write` adds 4,096 bytes to the file, whereas the last one adds 2,048 bytes to the file. From the recovered functions called by F_{write}, we find that F_{write} calls `write` 1,000 times, and each function call to `write` adds 256 bytes to the file. Since F_{fwrite} uses less function calls to complete the same task, it has a better performance than F_{write}.

Biased Measurement Results from Emulator: The overhead introduced by emulator-based app performance profiling tools [92] leads to biased measurement results. To evaluate the noise brought by the emulator to the performance measurement, we re-run the app 10 times on the emulator provided by Android Studio [36] and calculate the execution time of the two JNI functions. More precisely, we call `gettimeofday` at the very beginning and the end of each JNI function to get the start time (T'_s) and end time (T'_e) of the JNI function. Then, we compute the execution time of the JNI function via T'_e - T'_s. We discover that the average execution time of F_{fwrite} and F_{write} is 45.67 and 387.52 milliseconds, which are almost 20x bigger than those calculated by NCScope. Such observation suggests that NCScope can help analysts collect accurate data for performance analysis of native code.

Answer to RQ5: NCScope can aid performance analysis on native code by collecting the accurate data with little noise.

6 THREAT TO VALIDITY

Threats to the external validity of NCScope come from three aspects.

First, due to the intrinsic problem of dynamic analysis, NCScope may not trigger all instructions of the apps under analysis, and thus it may miss the behaviors that have not been executed. To mitigate this problem, we will apply automated app testing tools (e.g., Sapienz [74], Stoat [85], Fax [97]) to drive the execution of apps. In addition, we can customize targeted execution tools (e.g., IntelliDroid [91]) to drive the app to execute native functions. Moreover, we only study the native code based self-protection and anti-analysis behaviors in 500 financial apps and 300 malicious apps, respectively. In future work, we will enlarge our dataset and extend NCScope to identify more behaviors of native code.

Second, NCScope may miss new self-protection and anti-analysis behaviors whose detection rules are not include. To mitigate this threat, in future work, we will collect or design more rules to detect new self-protection methods and anti-analysis mechanisms.

Third, apps might detect NCScope via side-channels to evade the analysis. In future work, we will carefully optimize the code of NCScope to eliminate such side-channels as many as possible.

7 RELATED WORK

Various work has been proposed to analyze native code of apps. Owning to the complexity of native code [68], there are only a few static analysis based approaches. JN-SAF [90] and George et al. [66] statically analyze the native code to discover data leakage, command execution, and JNI reflection. LibRARIAN [55] statically builds libraries' identities by extracting strings in their native code. ATADetector [58] detects DBD, EMD, and DFD methods in native code. Due to the limitation of static analysis in handling dynamically loaded code, analysts may resort to dynamic analysis based tools.

However, existing dynamic app analysis tools have limitations on analyzing native code. Specifically, since DroidScope [98], CopperDroid [87], AndroidPerf [92], NDroid [93], and others [54] are built upon QEMU, they cannot analyze the anti-analysis apps with EMD behaviors. TaintDroid [64], TaintART [86], ARTist [57], and MERCIDroid [71] introduce significant overhead as they perform heavy-weight operations (e.g., information flow tracking) on apps, and thus they cannot analyze the anti-analysis apps with TCK behaviors. DroidTrace [99] and Malton [95] use `ptrace` and the DBI framework Valgrind [76], respectively, to analyze apps. Therefore, they are impeded by DBD or DFD behaviors. Ninja [77] and Happer [94] use ETM to trace apps and hardware breakpoints to retrieve memory data. However, neither of them is designed to profile behaviors of native code and they do not recover data flow. Moreover, due to the limited number of breakpoints, they have limitations in retrieving memory data.

Different from existing approaches, NCScope leverages ETM and eBPF to collect data and employs new methods to identify behaviors of native code. Thus, it will neither be hindered by DBD, EMD, DFD nor suffer from the problems incurred by the limitation of hardware breakpoints. Since NCScope introduces very low additional overhead, it can inspect the apps with TCK.

8 CONCLUSION

To analyze apps with native code, we propose NCScope, a novel hardware-assisted analyzer that collects execution traces via ETM and relevant memory data via eBPF and is equipped with new methods to inspect native code with very low additional overhead. Using NCScope, we conduct systematic studies on native code based self-protection methods and anti-analysis methods in financial apps and malicious apps, respectively, and observe that financial apps are not well protected but malware adopts various anti-analysis methods. Moreover, NCScope can aid memory corruption bug diagnosis and performance analysis for apps' native code.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful comments. This research is partially supported by the Hong Kong RGC Project (No. PolyU15224121), the National Natural Science Foundation of China (NSFC) under Grant U21A20464, the National Science Foundation under Grant (No. 1951729, 1953813, and 1953893), the Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006.

REFERENCES

[1] 2021. adeb. https://github.com/joelagnel/adeb.
[2] 2021. Advanced Debugging with ETM. http://ww1.microchip.com/downloads/en/AppNotes/Atmel-44045-32-bit-Cortex-M7-Microcontroller-Advanced-Debugging\-SAM-V71-V70-E70-S70-MCUs-with-ARM-ETM_App-Note.pdf.
[3] 2021. anti-emulator. https://github.com/strazzere/anti-emulator.
[4] 2021. AntiDebugandMemoryDump. https://github.com/darvincisec/AntiDebugandMemoryDump.
[5] 2021. Arm DS-5. https://developer.arm.com/tools-and-software/embedded/legacy-tools/ds-5-development-studio.
[6] 2021. ARM DSTREAM User Guide. https://static.docs.arm.com/dui0481/j/DUI0481J_arm_ds5_arm_dstream_user_guide.pdf.
[7] 2021. ARM PTM decoder, and ARM ETMv4 decoder. https://github.com/hwangcc23/ptm2human.
[8] 2021. Best Smartphones February 2021. https://benchmarks.ul.com/compare/best-smartphones?redirected=true.
[9] 2021. Bienvenido a Internet. https://f-droid.org/en/packages/org.bienvenidoainternet.app/.
[10] 2021. BPF Compiler Collection (BCC). https://github.com/iovisor/bcc.
[11] 2021. BPF Documentation. https://www.kernel.org/doc/html/latest/bpf/index.html.
[12] 2021. CVE-2019-11932. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11932.
[13] 2021. Debugging with GDB. https://ftp.gnu.org/old-gnu/Manuals/gdb/.
[14] 2021. Difference between fopen/open, read/write and fread/fwrite. https://topic.alibabacloud.com/a/difference-between-fopenopen-readwrite-and-freadfont-colorredfwritefont_8_8_31848651.html.
[15] 2021. Difference between ROOT and Custom ROM. https://forum.xda-developers.com/t/difference-between-root-and-custom-rom.3437151/.
[16] 2021. Embedded Trace Macrocell Specification. https://static.docs.arm.com/ihi0064/d/IHI0064D_etm_v4_architecture_spec.pdf.
[17] 2021. Extending the Kernel with eBPF. https://source.android.com/devices/architecture/kernel/bpf.
[18] 2021. F-Droid. https://www.f-droid.org/.
[19] 2021. Frida. https://frida.re/.
[20] 2021. Geekbench. https://www.geekbench.com/.
[21] 2021. IDA Pro. https://www.hex-rays.com/products/ida/.
[22] 2021. ISO/IEC 9899:TC2, Undefined behavior. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.
[23] 2021. Juliet Test Suite for C/C++. https://samate.nist.gov/SRD/testsuite.php.
[24] 2021. Juno Development Board. https://developer.arm.com/tools-and-software/development-boards/juno-development-board.
[25] 2021. Kernel Probes (Kprobes). https://www.kernel.org/doc/Documentation/kprobes.txt.
[26] 2021. KingRoot. https://www.kingroot.net.
[27] 2021. Malloc Hooks. https://android.googlesource.com/platform/bionic/+/master/libc/malloc_hooks/README.md.
[28] 2021. Mobile App Secure Development Guidelines. https://www.im.taichung.gov.tw/ash/documents/04_1.pdf.
[29] 2021. Mobile App Testing Guide. https://mobile-security.gitbook.io/mobile-security-testing-guide/.
[30] 2021. oatdump. https://android.googlesource.com/platform/art/+/kitkat-dev/oatdump/oatdump.cc.
[31] 2021. objdump. https://developer.android.com/ndk.
[32] 2021. OWASP Mobile Security Testing Guide. https://mobile-security.gitbook.io/mobile-security-testing-guide.
[33] 2021. PT_REGS_PARM* macros. arch/arm/include/asm/ptrace.h.
[34] 2021. Root Inspector. https://github.com/devadvance/rootinspector.
[35] 2021. RootBeer. https://github.com/scottyab/rootbeer.
[36] 2021. Run apps on the Android Emulator. https://developer.android.com/studio/run/emulator.
[37] 2021. Security of Mobile Payments and Digital Wallets. https://www.enisa.europa.eu/publications/mobile-payments-security.
[38] 2021. SignatureChecker. https://github.com/GarlicDipping/SignatureChecker-Android.
[39] 2021. Superser. https://superuserapk.org.
[40] 2021. SuperSU. https://github.com/khadas/supersu.
[41] 2021. System and kernel security for Android. https://source.android.com/security/overview/kernel-security.
[42] 2021. The native and Java heaps. https://www.ibm.com/docs/en/sdk-java-technology/8?topic=usage-native-java-heaps.
[43] 2021. Timing Checks. https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering#timer-checks.
[44] 2021. A tool for reverse engineering Android apk files. https://ibotpeaches.github.io/Apktool/.
[45] 2021. Uprobe-tracer: Uprobe-based Event Tracing. https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt.
[46] 2021. Using ftrace. https://source.android.google.cn/devices/tech/debug/ftrace.
[47] 2021. Using the Linux Kernel Tracepoints. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt.
[48] 2021. Whitepaper: Samsung Knox Security Solution. https://images.samsung.com/is/content/samsung/p5/global/business/mobile/SamsungKnoxSecuritySolution.pdf.
[49] 2022. CWE-415: Double Free. https://cwe.mitre.org/data/definitions/415.html.
[50] 2022. CWE-416: Use After Free. https://cwe.mitre.org/data/definitions/416.html.
[51] 2022. Java Native Interface Specification. https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html.
[52] 2022. Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide.
[53] 2022. mprotect. https://man7.org/linux/man-pages/man2/mprotect.2.html.
[54] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupe´, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. 2016. Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy. In *Proc. NDSS*.
[55] Sumaya Almanee, Arda Unal, Mathias Payer, and Joshua Garcia. 2021. Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code. In *Proc. ICSE*.
[56] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. PLDI*.
[57] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. 2017. ARTist: The Android runtime instrumentation and security toolkit. In *Proc. IEEE EuroS&P*.
[58] Stefano Berlato and Mariano Ceccato. 2020. A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps. *Journal of Information Security and Applications* 52 (2020), 102463.
[59] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. 2020. An Empirical Assessment of Security Risks of Global Android Banking Apps. In *Proc. ICSE*.
[60] Sen Chen, Ting Su, Lingling Fan, Guozhu Meng, Minhui Xue, Yang Liu, and Lihua Xu. 2018. Are Mobile Banking Apps Secure? What Can Be Improved?. In *Proc. ESEC/FSE*.
[61] Yunlan Du, Zhenyu Ning, Jun Xu, Zilong Wang, Yueh-Hsun Lin, Fengwei Zhang, Xinyu Xing, and Bing Mao. 2020. HART: Hardware-assisted Kernel Module Tracing on Arm. In *Proc. ESORICS*.
[62] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and X Wang. 2018. Things you may not know about android (un)packers: a systematic study based on whole-system emulation. In *Proc. NDSS*.
[63] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. 2003. Characterizing and predicting program behavior and its variability. In *Proc. PACT*.
[64] 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. *ACM Transactions on Computer Systems* 32, 2 (2014).
[65] Luca Falsina, Yanick Fratantonio, Stefano Zanero, Christopher Kruegel, Giovanni Vigna, and Federico Maggi. 2015. Grab'n run: Secure and practical dynamic code loading for android applications. In *Proc. ACSAC*.
[66] George Fourtounis, Leonidas Triantafyllou, and Yannis Smaragdakis. 2020. Identifying Java calls in native code via binary scanning. In *Proc. ISSTA*.
[67] Muhui Jiang, Tianyi Xu, Yajin Zhou, Yufeng Hu, Ming Zhong, Lei Wu, Xiapu Luo, and Kui Ren. 2012. EXAMINER: Automatically Locating Inconsistent Instructions between Real Devices and CPU Emulators for ARM. In *Proc. ASPLOS*.
[68] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. 2020. An Empirical Study on ARM Disassembly Tools. In *Proc. ISSTA*.
[69] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi. 2013. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Communications* 73, 4 (2013), 1421–1437.
[70] Ansgar Kellner, Micha Horlboge, Konrad Rieck, and Christian Wressnegger. 2019. False Sense of Security: A Study on the Effectivity of Jailbreak Detection in Banking Apps. In *Proc. Euro S&P*.
[71] Taehun Kim, Hyeonmin Ha, Seoyoon Choi, Jaeyeon Jung, and Byung-Gon Chun. 2017. Breaking Ad-Hoc Runtime Integrity Protection Mechanisms in Android Financial Apps. In *Proc. Asia CCS*.
[72] Bodong Li, Yuanyuan Zhang, Juanru Li, Wenbo Yang, and Dawu Gu. 2018. AppSpear: automating the hidden-code extraction and reassembling of packed android malware. *Journal of Systems and Software* 140 (2018), 3–16.
[73] Li Li, Tegawende Bissyande, and Jacques Klein. 2020. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering* (2020).

[74] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proc. ISSTA*.
[75] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. 2016. Straight-taint: Decoupled offline symbolic taint analysis. In *Proc. ASE*.
[76] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *Proc. ACM PLDI*.
[77] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM. In *Proc. USENIX Security*.
[78] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proc. NDSS*.
[79] Shisong Qin, Chao Zhang, Kaixiang Chen, and Zheming Li. 2021. iDEV: exploring and exploiting semantic deviations in ARM instruction processing. In *Proc. ISSTA*.
[80] Zhengyang Qu, Shahid Alam, Yan Chen, Xiaoyong Zhou, Wangjun Hong, and Ryan Riley. 2017. Dydroid: Measuring dynamic code loading and its security implications in android applications. In *Proc. DSN*.
[81] 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 *Proc. MobiSys*.
[82] Onur Sahin, Ayse K Coskun, and Manuel Egele. 2018. Proteus: Detecting android emulators from instruction-level profiles. In *Proc. RAID*.
[83] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Proc. S&P*.
[84] Lina Song, Zhanyong Tang, Zhen Li, Xiaoqing Gong, Xiaojiang Chen, Dingyi Fang, and Zheng Wang. 2017. AppIS: Protect Android Apps Against Runtime Repackaging Attacks. In *Proc. ICPADS*.
[85] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proc. ACM FSE*.
[86] Mingshen Sun, Tao Wei, and John Lui. 2016. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proc. ACM CCS*.
[87] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proc. NDSS*.
[88] Timothy Vidas and Nicolas Christin. 2014. Evading Android Runtime Analysis via Sandbox Detection. In *Proc. ACM AsiaCCS*.
[89] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep ground truth analysis of current android malware. In *Proc. DIMVA*.
[90] 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 *Proc. ACM CCS*.
[91] Michelle Y Wong and David Lie. 2016. Intellidroid: a targeted input generator for the dynamic analysis of android malware. In *Porc. NDSS*.
[92] Lei Xue, Chenxiong Qian, and Xiapu Luo. 2015. Androidperf: A cross-layer profiling system for android applications. In *Proc. IWQoS*.
[93] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. 2019. NDroid: Toward Tracking Information Flows Across Multiple Android Contexts. *IEEE Transactions on Information Forensics and Security* 14, 3 (2019), 814–828.
[94] Lei Xue, Hao Zhou, Xiapu Luo, Yajin Zhou, Yang Shi, Guofei Gu, Fengwei Zhang, and Man Ho Au. 2021. Happer: Unpacking Android Apps via a Hardware-Assisted Approach. In *Proc. S&P*.
[95] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *Proc. USENIX Security*.
[96] Carter Yagemann, Matthew Pruett, Simon P Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *Proc. USENIX Security*.
[97] Jiwei Yan, Hao Liu, Linjie Pan, Jun Yan, Jian Zhang, and Bin Liang. 2020. Multiple-Entry Testing of Android Applications by Constructing Activity Launching Contexts. In *Proc. ICSE*.
[98] Lok-Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proc. USENIX Security*.
[99] Min Zheng, Mingshen Sun, and John CS Lui. 2014. DroidTrace: a ptrace based Android dynamic analysis system with forward execution capability. In *Proc. IWCMC*.
[100] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proc. CODASPY*.
[101] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In *Proc. NDSS*.