Detecting Android Root Exploits by Learning from Root Providers

Ioannis Gasparis
University of California, Riverside
ioannis.gasparis@email.ucr.edu

Zhiyun Qian
University of California, Riverside
zhiyunq@cs.ucr.edu

Chengyu Song
University of California, Riverside
csong@cs.ucr.edu

Srikanth V. Krishnamurthy
University of California, Riverside
krish@cs.ucr.edu

Abstract

Malware that are capable of rooting Android phones are arguably, the most dangerous ones. Unfortunately, detecting the presence of root exploits in malware is a very challenging problem. This is because such malware typically target specific Android devices and/or OS versions and simply abort upon detecting that an expected runtime environment (*e.g.,* specific vulnerable device driver or preconditions) is not present; thus, emulators such as Google Bouncer fail in triggering and revealing such root exploits. In this paper, we build a system *RootExplorer*, to tackle this problem. The key observation that drives the design of *RootExplorer* is that, in addition to malware, there are legitimate commercial grade Android apps backed by large companies that facilitate the rooting of phones, referred to as root providers or one-click root apps. By conducting extensive analysis on one-click root apps, *RootExplorer* learns the precise preconditions and environmental requirements of root exploits. It then uses this information to construct proper analysis environments either in an emulator or on a smartphone testbed to effectively detect embedded root exploits in malware. Our extensive experimental evaluations with *RootExplorer* show that it is able to detect all malware samples known to perform root exploits and incurs no false positives. We have also found an app that is currently available on the markets, that has an embedded root exploit.

1 Introduction

Android is currently the most popular mobile operating system in the world, with 1.4 billion users worldwide and 87.5% of the market share [65]. Google, contrary to Apple (wrt iPhone), do not have complete control over either the hardware or the software of Android phones. On the positive side, this allows many hardware and other third-party vendors to build a competitive, customized, and diverse ecosystem. But on the other hand, the diversity of Android devices also introduces security issues. First, the OS update process varies from vendor to vendor (some are faster than others). For example, at the time of writing, only 29.6% of Android devices on the market have Marshmallow [40], which was introduced nearly 2 years ago. Second, the vendor customization of Android often introduces vulnerabilities at different levels of the software stack including application, OS kernel, and drivers [81, 85, 75, 7, 82]. Consequently, millions of users are exposed to various critical security vulnerabilities that plague such customized, typically older and unpatched devices [19, 27, 68].

Among all vulnerabilities, arguably the most pernicious are privilege escalation vulnerabilities that would allow attackers to obtain the root privilege – the highest privilege on Android. Such attacks are usually referred to as *root exploits*. Once it has acquired the root privilege, an attacker/malware can bypass the Android sandbox, perform many kinds of malicious activities, and even erase evidence of compromise. For this reason, malware with embedded root exploits are on the rise. Indeed, as apparent in recent news, it has become more and more common that malware found in third party Android markets or even in the official Google Play store, contain root exploits. For instance, in June 2016, Trend Micro reported GODLESS [55] , an Android malware family that uses multiple root exploits to target a variety of devices, affecting over 850,000 devices that were running Android 5.1 or earlier, worldwide. One month later (July), another Android malware dubbed HummingBad was reported to have infected more than 85 million devices and was found in 46 different applications, 20 of which were found on Google Play [45]. In September 2016, a Pokemon Go Guide app spotted in Google's Play Store, was found to contain root exploits as well [63]; the app had accumulated over 500,000 downloads by the time it was spotted and taken down. Considering that Google has already deployed a cloud-based app vetting

service viz.,"Google Bouncer" [39], these repeated instances demonstrate that it is both important and challenging to detect malware that carry out root exploits.

An even more concerning fact is that the number of newly discovered privilege escalation vulnerabilities (*e.g.,*, kernel vulnerabilities) is also on the rise [2]. Many of such vulnerabilities, such as DirtyCow [27], can even be used to root the latest versions of Android. So it is simply a matter of time before they are leveraged by malware to attack (potentially a large number of) unpatched devices.

In this paper, we aim to tackle the challenging problem of detecting malware that employ a variety of root exploits. The key observation that drives our approach is that, in the Android world, it is not just the malware that carry root exploits. There are legitimate and popular Android applications, often called root providers or one-click root apps, that root phones on behalf of users [81]. Many of these apps are commercial-grade and backed by large companies such as Tencent, Qihoo, and Baidu. They are capable of rooting tens of thousands of different Android devices using a hundred or more root exploits [81]. Note that rooting (as well as jailbreak) is considered legal [21], and users do want to root their phones to remove bloatware or unlock new features that were otherwise not available. These root exploits can serve as valuable resources towards aiding detection since they are highly customized (towards specific devices), reliable, and more importantly are likely to be used as is, by malware developers (discussed later). This means we can take advantage of these exploits to build a system (*RootExplorer*) that automatically extracts signatures from root exploits, and use those signatures for runtime malware detection.

Unfortunately, this seemingly simple strategy is not easy to realize in practice. The big obstacle is that almost all exploits are tailored towards specific Android devices, models, and/or OS versions. Screening apps in an emulator is unlikely to trigger and reveal the exploit, unless the environment matches exactly what the exploit expects. This in turn means that one may need tens of thousands of real Android devices to cover just all *known* root exploits. To overcome this obstacle, *RootExplorer* also learns the environment requirements from the aforementioned commercial root exploits and uses this knowledge to create the "expected" runtime environment so that it is capable of interacting with the exploits to drive their execution (*e.g.,* by pretending that a particular vulnerable device exists).

We design, prototype and extensively evaluate *RootExplorer* to detect root exploits present in malware. It consists of (a) an offline training phase where it extracts useful information about root exploits from one-click root apps using behavior analysis, and (b) an online detection phase where it dynamically analyzes apps in specially tailored environments to detect root exploits. We test our prototype with a large set of benign apps, known malware, and apps from third-party app marketplaces. Our evaluations show that *RootExplorer* yields an almost perfect true positive rate with no false positives. *RootExplorer* also found an app that is currently available on the markets, that contains root exploits.

In summary, the contributions are as follows:

- We identify and address the fundamental challenge of detecting Android root exploits that target a diverse set of Android devices. In particular, we learn from commercial one-click root apps which have done the "homework" for us with regards to (a) what environmental features are sought and (b) what pre-conditions need to be met, for a root exploit to be triggered.
- We design and implement *RootExplorer*, a fully automated system that uses the learning from commercial one-click root apps to detect malware carrying root exploits. Specifically, in an offline phase, it conducts extensive static analysis to understand the precise environment requirements and the attack profile of the exploits. It then utilizes the learned information to construct proper analysis environments and detects attempted exploits.
- We evaluate *RootExplorer* via extensive experiments and find that it can successfully detect all known malware that contain root exploits, including very recently discovered exploits and the ones that are used in other one-click root apps; *RootExplorer* results in no false positives with our test set. Using *RootExplorer*, we also find an app which is currently available on an Android market, that contains root exploits.

2 Background & Related Work

2.1 Root Exploits and One-Click Root Apps

As mentioned, one-click root apps are very popular among users and they are competing against each other to be able to root more phones and offer more reliable results. One of the reasons that companies develop these apps is that they also develop security apps or app management tools that also require the root privilege to function correctly (*e.g.,* antivirus software must have higher privileges than any malware [13]);

Interestingly, the competition between these one-click root apps have driven them to include the most comprehensive and advanced root exploits. For example, in 2015, it was reported that there are 39 families of directly usable root exploits that can be found publicly (with source code or binaries); in contrast, there were

59 families of root exploits found in a popular commercial one-click root apps, including exploits against publicly unknown or zero-day vulnerabilities [81], and exploits that can bypass advanced defense mechanisms like SELinux [41], Verified Boot [42], etc. On the contrary, although researchers have detected several malware families with root exploits, none of them contain previously unknown exploits [86]. We believe this is because most malware authors, except the so-called state sponsored, do not have the capability to develop new root exploits; hence, they typically only embed exploits that are developed by others (*e.g.,* one-click root apps).

While detecting malicious behaviors has been the focus of many prior efforts in the literature, detecting Android root exploits faces unique challenges. One such challenge is that specific preconditions (*e.g.,* environment constraints) need to be satisfied in order for such exploits to be triggered; this is hard because of the highly fragmented Android hardware and software. Specifically, not only do different phones have different device(s) and corresponding driver(s), even with respect to a universal kernel vulnerability such as the futex bug [31], the root exploit has to be tailored for different phones. This is because the actual kernels on different phones are different (*e.g.,* each has a different memory layout). As a matter of fact, one commercial one-click root app contains 89 different exploit payloads for the same underlying futex bug [81]. Consequently, malware carrying root exploits typically have specific environment checks to determine (1) what kinds of vulnerabilities are available and (2) how the attack should be launched. Thus, in order to detect a root exploit, an analysis environment must satisfy the necessary preconditions.

We categorize these preconditions into two corresponding types: (1) *environment checks* and (2) *preparation checks*. Environment checks gather information with regards to the environment such as the device type, model, and operating system versions. For instance, many times a particular malware will check whether it has a matching exploit for the current environment. If so, the specific exploit is selected from either a set of local exploits or a remote exploit database. This process is in fact also used by one-click root apps [81]. Preparation checks verify that the interactions with the underlying operating system are as expected, (*e.g.,* a vulnerable device file exists on the system and the driver returns expected results in response to specific commands). The number of preparation checks can be large, depending on the nature and complexity of the root exploits. This makes it difficult to manually prepare the right environment for each root exploit and detect them.

2.2 Android Malware Analysis

A relatively large chunk of Android related literature, is on malware analysis and malicious behavior detection. However, most of this literature focuses on detecting malicious behaviors like leaking/stealing private information and financial charging [86]. Unfortunately, no existing work tackles root exploit detection. We roughly categorize such work into three types: static analysis, dynamic analysis, and hybrid analysis.

Static Analysis: Static analysis is used to analyze an Android app's byte code and/or native code without running it inside an emulator or a real device. To detect information/privilege leaks, a set of tools [52, 10, 59, 50, 72, 18, 43] has been developed to perform information-flow analysis. Another popular direction is to model and detect malicious behaviors that are unique to Android. Pegasus [20] uses "Permission Event Graphs" to detect sensitive operations performed without the user's consent. Apposcopy [29] uses "Inter-Component Call Graphs" to detect Android malware. AppContext [79] uses contextual information (UI events and environmental triggers) to check access to sensitive operations. The advantage of static analysis is coverage and efficiency; it may however face problems when analyzing apps with heavy obfuscation. In fact, it has been shown that simple obfuscation techniques or transformations applied to known malware samples can often easily evade static detection by anti-virus software [62].

Dynamic Analysis: Dynamic analysis analyzes an Android app by running it inside an emulator or a real device. Similar to static analysis, many dynamic malware analysis systems also focus on information flow analysis and leak detection [28, 61, 83]. Others use system calls to model and detect malicious behaviors [17, 25, 78, 66]. Because malware can detect that it is being run in an instrumented environment such as an Android emulator [60, 46, 69], researchers have also proposed building sandboxes on real devices [15, 12] for this purpose. Dynamic analysis can usually overcome obfuscation techniques employed by malware, but a malicious behavior can only be detected if it is executed during the analysis. To overcome this, tools have been developed to systematically exercise the functionality of an app in the hope of triggering its malicious behaviors [11, 74].

Hybrid Analysis: Hybrid analysis can be divided into two categories. The first category combines static and dynamic characteristics to detect malicious behaviors [87, 76, 73]. The other category utilizes static analysis to guide dynamic analysis [84, 80, 11, 74].

2.3 Attack Modeling and Detection

Previous papers on attack modeling and detection mainly focus on filtering remote exploits like those launched by worms [23, 48, 58, 64, 51, 71, 24, 22]. Similar to those systems, *RootExplorer* also leverages program analysis techniques like symbolic execution to extract the attack signature. However, there are a few differences. First, due to fragmentation of the Android ecosystem, we do not always have the targeted device, *i.e.,* we need to derive both the attack signature and the corresponding environment requirements *without* the corresponding target system. Note that in aforementioned systems, in contrast, analysis is usually performed over the targeted software. Second, for remote attacks, the malicious payload usually contains shellcode; however, in local privilege escalation attacks, shellcode is rarely used – `ret2usr`, `ret2dir`, or direct kernel object modification (DKOM) are more common. Finally, due to polymorphic or metamorphic payloads, finding a good balance between false negatives and false positives is very challenging for network filters. Android root exploits are more difficult to morph (as shellcode is not part of the payload); more importantly, even though it is possible to generate polymorphic exploits, as previously discussed, most Android malware authors are not capable of doing so. For these reason, we decide to pursue our current approach, *i.e.,* derive system-call-based signatures purely from known exploits.

2.4 Other Related Work

Android Emulator Evasion: Recent works have shown how easy it is for malware authors to evade the Android emulator. Petsas *et al.* [60] apply three different detection heuristics and manage to detect most Android dynamic analysis tools. Vidas *et al.* [69] derive four different techniques based on differences in behavior, performance, hardware and software components and show how they can easily detect existing malware scanner tools that are based in emulators. Morpheus [46] is a system that can create up to 10,000 different detection heuristics for Android emulators. As a countermeasure, researchers [56] have begun to use real phones instead of emulators to analyze malware. We design our solution to be operable on both real Android devices and emulators, thereby making this issue orthogonal to our work.

Syscall-based Behavior Modeling: *RootExplorer* uses system-call-based behaviors to model and detect root exploit attempts. Syscall-based behavior modeling has been widely used to model and detect malicious behaviors [49, 14]. Our model is derived from the behavior graph proposed in [49], with adjustments to fit our scenario.

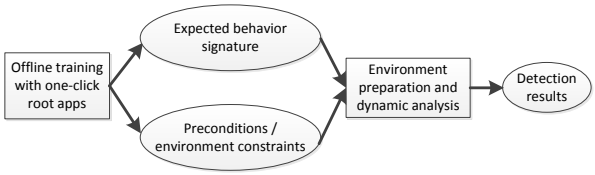

Figure 1: System overview

3 Threat Model and Problem Scope

The goal of *RootExplorer* is to detect Android apps that carry *root exploits*. Detecting other malicious behaviors is out-of-scope of this work and has been covered by many previous papers (§2). We also do not attempt to understand what the malware will do after acquiring the root privilege; we defer such an analysis to future work.

We envision our system to operate in the cloud (similar to Google Bouncer [39]), and that it will scan apps by dynamically executing the samples on real Android devices and/or emulators. For this reason, we restrict the source of the analyzed apps to be either from the official Google Play Store or from third-party marketplaces. We do not consider malware involved in targeted attacks such as APTs.

We assume that malware carrying root exploits can be obfuscated to prevent static analysis, and may be equipped with common anti-debugging/anti-virtualization techniques to detect the analysis environment. They may also download root exploits dynamically from a C&C server only when the desired Android device is detected. For triggering root exploits, we focus on understanding and providing the environment expectations. However, we do not handle malware that depends on specific user inputs (*e.g.,* passing a game level) to trigger the root exploit. We believe generating such inputs is orthogonal to this work and has been covered by other projects [11, 74].

Finally, we focus on detecting root exploits against known vulnerabilities; detecting unknown or zero-day exploits is out of scope of this work. We believe this is a reasonable limitation as no malware that has propagated through app marketplace has been found to contain zero-day exploits.

4 *RootExplorer* Overview

Figure 1 depicts the operations of *RootExplorer*. There are two key phases: (1) an offline training phase (static analysis) that extracts useful information about root exploits from one-click root apps and, (2) a detection phase (dynamic analysis) that dynamically analyzes apps in specially tailored environments to detect root exploits.

During training, we gather information about as many

different root exploits as possible. Since root exploits target *specific* devices, it is not possible to trigger all of their behaviors without proper environments. We thus resort to static analysis. For each exploit, we collect (1) sequence and dependencies of system calls that can lead to a compromise of the device, *i.e.,* behavior signature [14, 49], and (2) preconditions for *deterministically* triggering the exploit.

The first step of our offline analysis is to identify a feasible execution path that leads to the success of the analyzed root exploit. We use guided symbolic execution to solve this problem. In particular, we symbolize all external "inputs" to each root exploit (binary) and aim to find a shortest feasible path from the entry to the marked successful end point. We build our prototype symbolic execution engine based on IDA pro, which is capable of handling all the instructions and libc functions that were encountered in the training set of exploit binaries.

From the feasible execution path, we extract the sequence of system calls and the dependencies across system calls from the output of symbolic execution as well. This information is then used to construct the behavior signature. Since we already collect constraints over what information needs to be returned from the system through system calls (*i.e.,* preconditions) during symbolic execution, we just consult an SMT solver to provide a concrete instance of satisfying preconditions. Both pieces of information (behavior signature and preconditions) feed directly to the dynamic analysis phase towards preparing the right environment and satisfying necessary preconditions, to trigger and thereby detect various root exploits.

For this purpose, besides utilizing root exploits from one-click root apps, we could in theory utilize the many exploits with PoC code available on the Internet, but they all come in different "sizes and shapes". Some contain source code but often hard code values in certain variables; this renders the exploit suitable only for a specific tested Android device. Some have binaries only, which are obfuscated to prevent direct reuse. Therefore, We choose to work with a popular one-click root app for the purposes of training. The benefits are multi-fold: (1) the quality of exploits is likely very good, as they are offered in commercial products (*e.g.,* they don't contain unnecessary steps, and are unlikely to crash the system); (2) there is a rich variety of exploits available (60 families of exploits in our evaluation); (3) the exploits packaged in the same one-click root app are likely to be obfuscated in similar ways, making it possible to de-obfuscate all exploits at once and conduct static analysis on them.

Learning the expected behavior signature: The behavior signature of an exploit is extracted by analyzing the de-obfuscated exploit binaries. While there are many possible models to construct malware signatures in general, we favor system call based behavior signatures; this is because root exploits interact with the operating system through system calls in unique ways to exploit vulnerabilities. To this end, we build our behavior signature largely based on prior work on extracting a malware behavior signature from system calls [14, 49]. This allows our dynamic analysis to keep track of the progress of an exploit and confirm it when all of its steps have been performed. More details are provided in §5.

Learning preconditions: As discussed earlier in §2, there are two types of preconditions that have to be satisfied with regards to a root exploit in general: environment related and exploit preparation related. Environment preconditions dictate whether the underlying Android device model and kernel version match what are expected by the exploit. After training, our dynamic analysis environment can provide the expected Android device information to trigger an exploit. Normally it is difficult to determine which exploits work against which Android devices (because one needs to ideally test an exploit against real devices). Fortunately, one-click root apps already provide this information to a large degree. Specifically, the one-click root app we studied downloads a different set of exploit binaries depending on the device information that is reported to its backend server. By reverse engineering their protocol, we have effectively built a mapping from a list of more than 20K Android device types (available from [1]) to their corresponding exploits. The assumption is that a one-click root app has a reasonably good idea of which exploits can target which device.

For exploit preparation related preconditions, we give the symbolic constraints collected along the feasible path and ask the SMT solver to construct a concrete satisfying instance such that when replayed during dynamic analysis, can deterministically trigger the analyzed root exploit. For instance, if an exploit expects to open a vulnerable device file successfully, the "input" to the exploit program is the return value of the `open()` syscall, which needs to take a non-negative value according to the symbolic execution. Once we learn such preconditions, our dynamic analysis environment can provide the same expected "input". We will present the detailed design of the symbolic execution framework in §6.

5 Behavior Graph Analysis

Since Android malware (especially those that contain root exploits) typically obfuscate their payloads heavily [86], dynamic analysis is the obvious choice over static analysis, for the purposes of detection. However, as discussed earlier, dynamic analysis wrt root exploits is difficult as such exploits target *specific* Android devices. Without the right environment, such exploits are likely to

terminate prematurely, thereby preempting detection.

To overcome this hurdle, we leverage de-obfuscated binaries from a one-click root app from our prior study [81] to extract the behavior signatures of root exploits. A behavior signature is constructed by abstracting the low-level operations into a high-level behavioral representation [49, 14]. One can check for malware samples that exhibit similar behaviors at runtime and thereby detect the presence of the particular exploits. In the case of root exploits, since they interact with the kernel (or device drivers) in unique ways to exploit an OS vulnerability, we choose to capture behaviors by modeling system call events. Instead of reinventing the wheel, we borrow the system call modeling technique from ANUBIS [49] with slight adjustments. Specifically, we follow the definition of "behavior graphs" [49] that are used to describe OS objects, system calls that operate on these objects and, relationships across system call events (*e.g.,* the result of one system call is used as a parameter on another system call).

The behavior graphs are directed acyclic graphs where the nodes represent objects and system calls, and the edges encode (1) the dependencies between objects and system calls, and (2) the dependencies across system calls. Compared to the traditional model of simply looking at a sequence of system calls [44], a behavior graph constrains the order of only dependent operations through an explicit edge (and never constrains independent operations).

While the high-level behavior graph is similar to that proposed in [49], we highlight the main differences here: (1) We statically extract the behavior graph instead of extracting it from a dynamic trace (as is done in ANUBIS). This leads to different requirements as elaborated later. (2) Since we target Android, the system calls are mostly inherited from Linux and are different from Windows.

5.1 Generating Training Behavior Graphs

We now describe how we automatically generate the behavior graph statically, by analyzing de-obfuscated ARM root exploit binaries [81]. The system call invocations, and their hard-coded arguments are generally easy to identify. This allows us to know what OS objects are created (*e.g.,* a file name), and how they are operated on (*e.g.,* Read-only or Read/Write). The main challenge that we face is to extract the dependencies across system calls.

Extracting data dependencies: To extract dependencies across system calls, we look for cases where the arguments for one system call is derived from a previous system call. Previous work [49] utilized taint analysis to derive such dependencies. In our system, since we perform static analysis over de-obfuscated binaries, we take a slight different approach. Specifically, when we use symbolic execution to find a feasible success path, we symbolize all the outputs of system calls. During the analysis, symbolic values are propagated along the execution path. To determine whether a path is feasible, whenever we meet a conditional branch that depends on symbolic value, we consult the solver to see if the corresponding path constraints are solvable. If we consider a symbolic value as tainted, then symbolic execution itself, already constructs the data dependencies between system calls, *i.e.,* if the input argument(s) of a system call is a symbolic value, then it must have a data dependency over one or more previous system calls. More importantly, the symbolic formulas of such input arguments also specify *how* they are depend on each other. Based on this observation, we extract the data dependencies between system calls by simply naming the symbolic values returned by system calls according to the system call names and their sequence in the feasible path (*e.g.,* `read2_buf`).

Extracting control dependencies: Symbolic execution does not directly provide control dependencies. To extract such information, we simply conduct a backward analysis. In particular, when outputting the feasible path discovered via symbolic execution, we also mark each control point that *directly* depends on the symbolic value with the system calls that introduced that value. Using the path, we start from the end point and traverse the trace backwards to look for system call invocations (e.g., `BL mmap`). Once we find a system call invocation, we can extract its control dependencies over previous system calls by searching for the closest "tainted" branch that precedes this syscall invocation. Alternatively, we could have used static binary taint analysis to extract both data and control dependencies.

Modeling of libc functions: The exploit binaries in our training set do not generally call the system calls directly (as typical with most native code). Instead, they call the libc functions (in Android, it is called Bionic). Fortunately, most are simply wrappers of system calls and have the same exact semantics. In cases they are not exactly the same, for example, `fopen()` vs. `open()`, we model the Bionic version `fopen()` by mapping its arguments and return values to `open()`. Furthermore, we leverage function summaries to model most encountered libc functions that need to be analyzed by symbolic execution.

5.2 Examples

Device Driver Exploit: To illustrate our behavior graph analysis, we consider a popular device driver exploit that targets the vulnerable Qualcomm camera driver, "camera-isp". This example is taken directly from our training data set from a popular one click root app. In

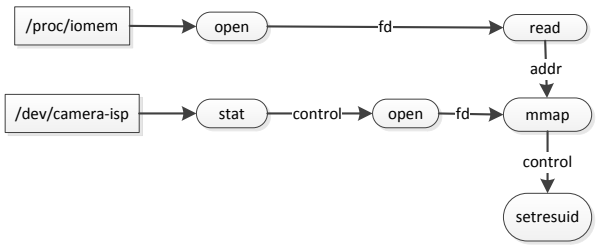

Figure 2: Behavior graph for the "camera-isp" exploit.

brief, the vulnerable device driver allows any program to map any part of the physical address space into the user space, which can subsequently allow the disabling of the permission check in `setresuid()` system call. This allows an attacker to change the running process into a root process.

Figure 2 represents the behavior graph. The exploit needs to open two separate files, the vulnerable device file `/dev/camera-isp` and the helper file `/proc/iomem` which has the information about where the kernel code is located in the physical address space. Both files are checked with the `open()` system call to ensure that they can be successfully opened. The device file is checked once more in the beginning, via a `stat()` system call, for existence. The exploit then attempts to `mmap()` the kernel code region into the user address space with read/write permissions; however, the exact offset (argument in `mmap()`) is retrieved from the read result of the `/proc/iomem`. After the `mmap()` is successful, the exploit searches for a particular sequence of bytes in the mapped memory that corresponds to the code blocks for `setresuid()`. Upon locating the code block, it patches the code block by writing to a specific offset, which effectively eliminates the security check in `setresuid()` (the above two steps are invisible in the behavior graph). Then the exploit simply calls `setresuid(0,0,0)` to change the uid of itself to root. Finally, as mentioned earlier, all exploit binaries in our training set, end the execution with a check through `getuid()` to verify that the exploit process has obtained root.

Note that due to space constraints, we do not annotate the graph with the exact arguments (*e.g.,* file open with a read/write permission or read-only). We also do not label whether the system call succeeded or not. In most exploits, all system calls need to be successful in order to compromise the system and typically the failure of a system call will immediately result in an abort.

Kernel Exploit: As a second example, we consider Pingpong root [77], one of the most recent generic root exploits that can target almost all Android devices released prior to mid-2015. The case also reflects one where the exploit creates multiple processes. In particular, the key exploit logic [35] is conducted in the main process, including `mmap()` at a specific address, and invoking multiple `connect()` calls on the same ICMP socket (we omit the complete behavior graph for brevity). In addition, One or more child processes are created as helpers to construct as many ICMP sockets as possible for padding. Since the `fork()` occurs in a loop (up to 1024 iterations), it is necessary for symbolic execution to identify and choose one feasible path. Specifically, the analysis output is that as long as the loop is executed once, a feasible exploit path can be constructed. This means that we can simply unroll the loop once and have a new behavior graph constructed for the child process (which is connected to the parent behavior graph via a `fork()` edge). Note that unrolling the fork loop more times is also feasible which will cause identical behavior graphs to be constructed. In this case, all behavior graphs will need to be matched so that we can claim an exploit is detected. It is worthwhile mentioning that the precondition analysis (which will be described in more detail in the next section) is conducted jointly, and will ensure that the first `fork()` will succeed at runtime, thus causing the exploit to match the behavior graph with one child process only.

5.3 Using Behavior Graphs in Detection

Once the behavior graphs for different root exploits are generated offline, we are able to use them for detection in a scanner (similar to Google Bouncer). More precisely, by monitoring system call invocations (and arguments), our dynamic analysis environment determines if the behavior of the program under analysis matches any of the learned behavior graphs. The matching algorithm is similar to that in [49]. We only briefly describe the procedure below and the design decisions that were made.

To find a match in the behavior graph, it is necessary to ensure the following: (1) The order of system calls conforms to the dependencies represented in the learned behavior graph. In addition, the dependencies in the behavior graph need to be maintained at runtime as well. This can be checked using dynamic taint analysis [14, 49]. (2) The exact values of the arguments for system calls match (*e.g.,* a file opened with read/write permission). For those arguments whose values cannot be determined statically during training, they will simply be considered as wildcard values that can match any value at runtime. (3) A system call's status (either success or failure) matches with the one in the learned behavior graph.

We observe that the root exploits typically have unique inputs to the system via system call arguments, which makes them easy to distinguish from legitimate programs. We therefore relax requirement (1) by only verifying simple dependencies at runtime (*e.g.,* a file `read()` depends on the output of `open()`). Such cases can be checked through the OS objects monitored in the ker-

nel, without conducting an expensive taint analysis. For more complex dependencies such as the values obtained through `read()` affecting a system call `mmap()` as shown in Figure 2, we only require that the order is the same as constrained on the graph, *i.e.,* `read()` happens before `mmap()`. We plan to implement the dynamic taint analysis for stricter dependency enforcement in future work. Alternatively to improve efficiency, we could also apply the optimization proposed by Kolbitsch *et al.* [49].

6 Satisfying Exploit Preconditions

It is crucial to build an environment that can satisfy the preconditions expected by root exploits. More importantly, because our behavior graph is constructed over one successful path, if an analyzed app contains root exploits, our dynamic analysis environment must deterministically coerce the app to follow that path, *i.e.,* the app must be made to reveal the same set of malicious behaviors that match the learned signature. This means that whenever the exploit asks the environment for certain results, we must return them as expected.

The problem naturally maps on to the common debugging and testing problem of generating the proper inputs to a program, so that it will reach a particular target statement [53, 26, 22]. Here the target statement is the end point of the root exploit, *e.g.,* the `getuid()` call. The "inputs" are the system call results, including (1) system call return values and, (2) other return results through arguments (*e.g.,* a buffer filled in `read()`). Our solution to this problem is symbolic execution. Specifically, we symbolize all the "inputs" from system calls and leverage symbolic execution to find the shortest feasible path that can reach the target instruction from the entry point. Once we find such a path, we then ask the SMT solver to generate a concrete instance of the inputs which will be "replayed" during dynamic analysis.

With respect to the system call return values, we consider two types of system calls: (1) Those that return a reference to kernel object, *e.g.,* `open()` and `socket()` return a file descriptor; and `mmap()` returns the address of the "memory-mapping object". (2) The remaining ones (*e.g.,* `stat()`) that return either 0 (indicating success) or an error code. For type (1), since file descriptors and mapped addresses are determined dynamically by the OS and the constraints are typically simple (just `!= 0`), we symbolize their return values as a Boolean during analysis and do not force a specific value during runtime. Instead, we simply choose to force a success or failure based on the Boolean and let the OS assign the concrete return value. To allow expected interactions with the corresponding kernel objects, we use "decoy objects" (explained later) instead of tracking those references. For type (2), we just symbolize their returned values normally as bit-vectors and ask the solver to generate a satisfying value.

```
fdIo = open("/proc/iomem");
// locate the kernel code offset in physical memory
while ((line = readline(fdIo)) > 0) {
   if((buf = strstr(line, "Kernel code")) != NULL) {
      addr = getAddress(buf);
      break;
   }
}

int getAddress(buf) {
   return atoi[buf-20];
}
```

Figure 3: Pseudo code of `proc/iomem` read

For system calls that return results through arguments, they are always pointers passed in user programs (*e.g.,* read buffer). We use these input pointers to symbolize the corresponding memory content. Going back to the first example exploit in §5.2, after reading from the file `/proc/iomem`, the exploit attempts to read the starting physical address of the kernel code. This procedure is illustrated in Figure 3. As we can see, the exploit reads the file line by line to look for the constant string "Kernel code". Once the line is located, it retrieves the kernel code base address (through the `getAddress()` call) at the -20 offset relative to the returned buffer of `strstr()`. There are effectively two loops in the program. The first is the `while` loop; the second is inside `strstr()`. In this particular case, the discovered feasible path says that the `while` loop can iterate just once, indicating that we can return the string containing "Kernel code" when the first line is retrieved using the `read()` system call. However, the feasible path also says that the loop in `strstr()` needs to iterate at least 20 times[1]; in other words, "Kernel code" needs to start at line [20]. This is because the `getAddress()` call reads the location at `buf-20`. If `buf` is at the beginning of line, then `buf-20` would be reading something out of bound.

In this case, the address returned from `getAddress()` is not further constrained later, which means that `line[0]` to `line[19]` are unconstrained and can take any value. Therefore, the constraint solver will generate an output for `line` with something like "abcdefghijklmnopqrstKernel code". Further, since the `read()` system call only reads one line, we will place the single line content into the expected file object. There is a similar case later on involving a search through the memory for constants after `mmap()`, which can be resolved similarly.

Decoy Objects: During dynamic analysis, we can provide the preconditions we learned by forcing/faking

[1] In our real implementation, we use function summary to handle all encountered external library calls.

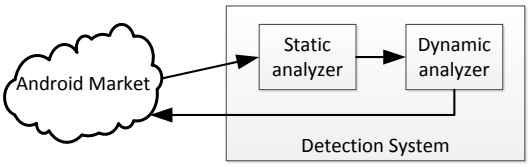

Figure 4: Operational model of the detection system

all syscall results. However, to improve the robustness of our environment (*i.e.,* making it more real), we decided to use decoy objects to provide expected results for operations over certain type of kernel objects. Doing so would allow us to "tolerate" certain operations (*e.g.,* `stats()` that are not observed during our offline learning phase.

Currently we only support three types of decoy objects: files, socket, and device drivers. These are created in two ways. If the target objects (such as a vulnerable device driver) do not exist in our analysis environment, we simply create decoys. If the objects (such as `/proc/iomem`) already exist in our environment, instead of opening the real file, we "redirect" the file open operation to the alternative decoy object so that we can return the expected content.

7 Detecting Root Exploits

Thus far, we have described the *training phase*, where we generate both the behavior graph and the environment constraints. In this section, we provide details about the components of our detection system (*testing phase*). We first present an overview of our system's operational model and then describe its components in detail.

7.1 Operational Model

As mentioned earlier, we envision *RootExplorer* to be used as an app vetting tool for Android markets. When a developer submits an app to the market via a web service, we envision the market pushing it to *RootExplorer*, as depicted in Figure 4. First, we employ a static analyzer (different from the static analysis during the training phase), which performs several checks to filter apps that are unlikely to contain root exploits (details later). Subsequently, it determines "with which kind of mobile device(s) or emulator(s)," the dynamic analysis will be performed. Upon completion of the dynamic analysis, the detector collects the results and determines if the app contains a root exploit and if so what exploit it is. If the app does have root exploits, it informs the Android market and saves the hash of the app to the central database; otherwise the app is moved either to a different malware scanner (*e.g.,* Bouncer) that is orthogonal to our system

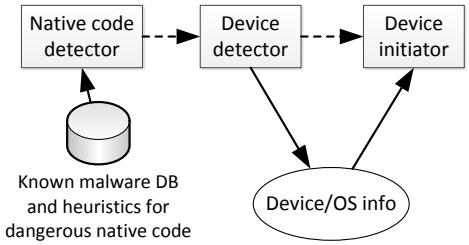

Figure 5: Static analyzer

or for publication in the Android market. The dynamic analyzer can be run on either real phones or Android emulators (or a mix of both), and can be easily integrated into various malware analysis environments as needed.

7.2 Static Analyzer

The static analyzer consists of three components as shown in Figure 5. The first component is the native code detector. Since almost all root exploits are written in native code (certainly the case for the one-click root app we study), it is natural to check whether the apps contain native code. Specifically, the native code detector does the following checks to filter apps that are extremely unlikely to contain root exploits: (1) Whether the app matches signatures of known malware samples that contain root exploits. If so, we abort any further analysis and flag the malware. (2) Whether it has any native code or has the capability of dynamically loading native code (*e.g.,* through the network). If negative, we can safely skip the analysis of this app. (3) If it contains native code, similar to prior work [8], we use a list of custom heuristics to decide if they can *possibly* contain root exploits (*e.g.,* whether any dangerous system calls are being called). If negative, we do not analyze the app further.

If the native code detector did not abort the analysis, the app is moved to the device detector. This is responsible for determining "under which environment the app should be dynamically analyzed." The observation is that since malware can embed different exploits targeting different Android devices, they usually contain logic that detects the type of the Android environment. Thus, we look for any such logic that performs checks against hard-coded device types.

The last component is the device initiator, which generates the Android environment based on the output of the device detector. We describe the device detector and the device initiator in more details below.

Device detector: This component parses the decompiled bytecode (using androguard [9]) and finds the methods (A) that contain code that check either the Android version that resides in the static class android.os.Build.VERSION, or the type of

the device that resides in the android.os.Build class, or the version of the Linux Kernel (*e.g.,* by `Runtime.getRuntime().exec(‘‘uname’’)` and reading the /proc/version file). Furthermore, it finds the methods (B) that either run an executable native file (*e.g.,* `Runtime.getRuntime().exec(‘‘/sdcard/foo’’))` or call a function in a native binary (*e.g.,* library files). If there is a program path from the methods that are members of (A) to the ones in (B), it finds which conditions should be satisfied and creates the appropriate Android environment. Similarly, the same procedure is performed in native code. In the case that the native code is obfuscated or even downloaded via a C&C server, the device detector simply picks a few popular candidate device types randomly, with the view that the malware will likely target one or more popular devices.

Device initiator: Android stores the device information in system files such as `/system/build.prop` and `default.prop`. `/system/build.prop` contains specific information about the mobile device such as the Android OS version, the name and the manufacturer of the device. In addition, there are also system files such as `/proc/version` and `/proc/sys/kernel/*` inherited from Linux that store information about the Linux kernel. When the system boots, the Android's property system reads the information from these files and caches them for future use. The device initiator monitors all such interfaces via which an app can learn about the device and obtain OS information. Since we have collected a database of Android devices from the online repository [1], we know what values to modify in the system files or what to return through the proc interfaces.

7.3 Dynamic Analyzer

The dynamic analyzer consists of two parts as illustrated in Figure 6 viz., a Loadable Kernel Module (LKM) and a background service process. The LKM hooks every available system call in the Android Linux Kernel. In addition, it creates a character device that can be opened by only the background service (to prevent malware from tampering with the communication), and with which a communication link is established between *user-land* and *kernel-land*. The LKM tracks only a specific app (under analysis) and its child processes at any moment in time. The background service stores the training models including behavior graphs and environment constraints, as well as the state of the current running app (*e.g.,* what part of a behavior graph has been matched and what environment constraints are supposed to be returned next).

Once a hooked system call is called by the app under analysis, the execution is directed to our LKM which then transmits a specially crafted message that contains the system call names as well as their arguments to the

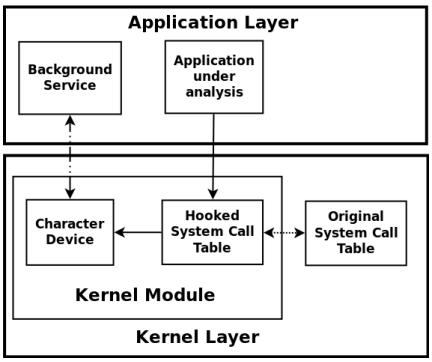

Figure 6: Dynamic Analyzer.

background service through the character device. Based on the behavior graph and environment constraints, the background service is responsible for deciding what action is going to be taken, and it returns that action to the LKM. First, it checks the behavior graph to see if the system call in question matches any new node. If not, it does nothing and simply instructs the LKM to execute the system call as is. If a new node is matched, it further checks if it is an object creation system call such as `open()` or `socket()`. If so, it deploys a decoy object to satisfy the environment constraints as described in §6. Otherwise, if it is a system call that operates on an existing object, the return results will be served from the data prepared ahead of time for the decoy object (*e.g.,* file content).

Note that deploying decoy objects has to be done carefully. As mentioned, Android root exploits often need to be adapted to work on different devices, even when they target the same underlying vulnerability. For instance, the device file `/dev/camera-isp` can be exploited slightly differently on different Android phones that all have the vulnerable device driver; this will cause slightly different behavior graphs and preconditions to be generated (*e.g.,* the input to a vulnerable device file will look different), and the expected return results from a system call may be different. Therefore, once we have decided to disguise as a particular Android device (*e.g.,* Samsung Galaxy S3), we will need to choose the behavior graphs and preconditions accordingly (obtaining such a Android device to exploit binary mapping is discussed in §4). Otherwise, the decoy objects we deploy may be for the wrong Android device which will in fact fail to detect the exploit.

8 Evaluation

In this section we describe the evaluations of *RootExplorer*. We focus on its effectiveness wrt the following aspects: (1) can it detect synthetic and real malware containing root exploits? (2) does it cause false alarms on benign apps? (3) does it miss malware samples?

8.1 Environment Setup

Training parameters: Our training database contains 168 different root exploits that were designed for different devices and were obtained from a commercial one click root app. The number of devices that we can successfully emulate currently based on the root exploit database is 211. We trained with all 60 families of root exploits.

Testing dataset: We have the following categories of apps for evaluation:

1. *Samples that are known to contain root exploits.* This includes publicly distributed exploit PoCs [38, 36, 34, 33] and GODLESS malware [32], and seven other one-click root apps (*different from the one we trained with*) which also contain many different root exploits.

2. *Samples that* may *contain root exploits.* We obtained a list of 1497 malware samples from an antivirus company, and have crawled 2000 recently uploaded apps between January and February 2017, from four unofficial Android app markets: 7723 [3], ANDROID life [4], MoboMarket [6] and EOEMARKET [5]. We target third-party markets because they are known to have more malware than the official Google Play [87].

3. *Samples that almost certainly* do not *contain root exploits.* This includes the top 1000 free apps from Google play. As they are extremely popular, it is very unlikely that they contain root exploits. This set is used to evaluate the false positives (if any) with *RootExplorer*.

Analysis Testbed: The experiments are performed on a Lenovo Laptop with a quad core Intel Core i7 2.00GHz CPU, 16GB of RAM, and a hard drive of 1TB at 5400 rpm. We use an Android emulator for analyzing the malware[2]. To make the emulator appear as realistic as possible, it is loaded with real files, such as music, pictures and videos. Furthermore, it contains a call log, SMS history and contacts, as well as various installed apps. We have modified the binary image of the emulator, in order to show that it has a real phone number and a real International Mobile Equipment Identity (IMEI) number. Finally, the `build.prop` file (containing the device information) is updated appropriately prior to each experiment. Each app is analyzed starting with a clean image of the emulator in order to avoid any side effects that a previously tested malware app can have on the emulator. A simple micro-benchmark on the `open()` system calls shows that the system call monitor increases the execution time of `open()` by 75%, on average.

Input generator: To achieve as much code coverage as possible when executing an app (in hope that root exploits will be triggered), we leverage DroidBot [37], a lightweight test input generator for Android that generates pseudo-random streams of user events such as clicks, as well as a number of system-level events. DroidBot can generate random events on its own, or generate events based on the manifest file of the app, or can take as input a file with predefined events. In our experiments, we use randomly generated events ("black-box" technique) and events based on the manifest file of the app ("gray-box" technique).

[2]Even though the system runs on real phones, we choose an emulator based approach since it is easier to run a large set of experiments concurrently.

One-Click App	Exploit
O_1	**/dev/camera-sysram**
O_2	**/dev/graphics/fb5**
O_3	**/dev/exynos-mem**
O_4	**/dev/camera-isp**
O_5	**/dev/camera-isp**
O_6	**/dev/camera-isp**
O_7	**towelroot**

Table 1: One-Click apps with the discovered exploits.

8.2 Effectiveness

We evaluate *RootExplorer* against all the test datasets mentioned earlier containing 4497 apps in total. Overall, we do not find any false positives, *i.e.,* benign apps are never mistakenly reported to contain root exploits. For the known malware samples, we obtain the ground truth either from the fact that github explicitly states that it is a root exploit, or via cross-validation with VirusTotal and the antivirus company that we work with. Out of 8 known malware families containing root exploits, we do not find any false negative. We describe the details below.

Unit testing: To obtain assurance that the training phase works as expected, We execute the 60 families of root exploits (from the training data) in our dynamic analysis environment and see if they can be detected. Note that this means that the training and testing data are exactly the same. If any of these exploits cannot be detected, it indicates that the behavior graphs or preconditions that were prepared are in fact incorrect. The testing results show that all of the exploits are successfully detected.

Detecting other one-click root apps: Since we have not trained *RootExplorer* with exploits from other one-click root apps, this test allows us to further confirm that the system works well. In particular, the exploits from these apps may or may not be implemented exactly in the same way as the ones in our training set, being able to trigger and detect them is a promising sign. Table 1 lists the first exploit that was caught upon running 7 other one-click root apps on an emulated Samsung Galaxy S3 device. Interestingly, different one-click root apps in fact choose to launch different exploits against our de-

Exploit	VirusTotal	RootExplorer
diag	1/57	✓
exynos	4/57	✓
pingpong	1/57	✓
towelroot	3/57	✓

Table 2: Detection rate for debug compilation.

Exploit	VirusTotal	RootExplorer
diag	0/57	✓
exynos	1/57	✓
pingpong	0/57	✓
towelroot	1/57	✓

Table 3: Detection rate for obfuscated compilation.

vice. For instance, with O_1, we caught an exploit related to the `/dev/camera-sysram` driver, while O_2 and O_3 triggered exploits against `/dev/graphics/fb5` and `/dev/exynos-mem` respectively. The results showcase the effectiveness of *RootExplorer* in detecting a wide variety of exploits.

Detecting Exploit PoCs from the Internet: We next take four proof-of-concept root exploits (with source code) that we can find on github [38, 36, 34, 33], and embed them in a testing Android app we build, that simply roots a phone upon touching a button. We first check the effectiveness of current anti-virus programs against the "malware" we built containing publicly available PoCs. We scan the app using the virusTotal API [70] which contains 57 anti-virus programs (*e.g.,* Trend Micro) capable of scanning Android apps . Table 2 shows the detection rates for the case where we compiled the source code with all the debug options on and without any obfuscation, while Table 3 shows the results when the compiled binaries are obfuscated using the O-LLVM obfuscator [47] and packed using UPX [67] (both are off-the-shelf tools).

In brief, without obfuscation, all four exploits can be detected by at least one antivirus. However, with simple obfuscation, only exynos (CVE-2012-6422) [30] and towelroot (CVE-2014-3153) [31] can be successfully detected and that too by only one antivirus. On the other hand, *RootExplorer*, by preparing the right preconditions and observing the exploit behaviors at runtime, can detect every exploit regardless of the obfuscation and packing methods.

Detecting GODLESS: GODLESS [55] is a family of malware that employs multiple root exploits, and has caused havoc in the wild since mid-2016. *RootExplorer* is extremely effective in detecting the exploits in the GODLESS malware family. Its source code is largely based on the open source repository on github [32]. Specifically, GODLESS checks the device type against a predefined, populated database of supported exploitable devices. Depending on which device it is running on, it invokes a corresponding, appropriate exploit. The process is iterative. It begins with exploit `acdb` and checks if the device is in the database, and only if so, will continue with the actual exploit. Upon failure, it moves on to next exploit which is `hdcp`, and so on until it has tested the last exploit viz., `diag`. We run GODLESS against 5 different emulated devices to showcase that *RootExplorer* is effective in properly stimulating the right exploit for a device. Table 4 shows the results (with the emulated devices). The exploits with code name `msm_camera`, `put_user` and `fb_mem` can be caught using any emulated device; this is because these exploits affect a large number of devices and seemingly, the author of GODLESS does not even know the complete list of devices they affect. Instead, GODLESS simply always tries to execute them without checking the actual device type. Of course, if a target device does not have the vulnerable device file such as `/dev/msm_camera`, the exploit will simply abort and the next exploit is attempted. Since *RootExplorer* is trained to prepare the preconditions for all the exploits at all times including `msm_camera`, it deploys the decoy file `/dev/msm_camera` on demand when GODLESS tries to open it, and can therefore always trigger and detect its complete malicious behavior with respect to this exploit.

Detecting Malware in the Antivirus malware dataset and 3rd-party Android Markets: For each app from the 1497 malware samples we received from an anti-virus company and the 2000 apps downloaded from four third-party Android markets, we apply *RootExplorer* for 10 minutes using Droidbot with an emulated Samsung S3 device; the kernel version, build ID, and the model of the device are set to 3.0.31-1083875, JZO54K, and GT-I9300 respectively. Upon booting the emulated device, Droidbot launches the main activity of each app and generates random touch events and system events such as `BOOT_COMPLETED` every second. Meanwhile, our tool runs in the background and analyzes all the system calls that the app uses. To measure the number of false positives and false negatives, we scan those apps with VirusTotal. Among all the apps, *RootExplorer* detected two true positives (and has no false positive).

The first app is named *Wifi Analyzer* from the MoboMarket [6], which was discovered to contain the pingpong root exploit [77] (`md5` ea1118c2c0e81c2d9f4bd0f6bd707538). At the time of writing, the app is still alive on the market. After consulting with VirusTotal and an antivirus company, we confirmed that it is an instance of the `rootnik` malware family [57]. We have reported to the market and are waiting for the app to be removed.

Another detected app is a Flashlight app from the Antivirus malware dataset, containing the `camera-isp` root exploit. It has an `md5` of 1365ECD2665D5035F3AB52A4E698052D.

	HTC J Butterfly	Fujitsu Arrows Z	Fujitsu Arrows X	Galaxy Note LTE	Samsung S3
acdb	✓	✗	✗	✗	✗
hdcp	✗	✓	✗	✗	✗
msm_camera	✓	✓	✓	✓	✓
put_user	✓	✓	✓	✓	✓
fb_mem	✓	✓	✓	✓	✓
perf_swevent	✗	✗	✓	✗	✗
diag	✗	✗	✗	✓	✗

Table 4: Emulated devices and corresponding exploits caught by *RootExplorer* in GODLESS malware.

Upon starting, the app tries to access the files `/system/xbin/su` and `/system/bin/su`. *RootExplorer* returns the appropriate errors to make the app believe that it is running on an un-rooted device. Interestingly, only when DroidBot delivers the `BOOT_COMPLETED` event to the app, the root exploit is triggered. In the beginning, it opens and reads the file `/proc/kallsyms` four times to retrieve the address of certain kernel symbols. After that, it opens the vulnerable `/dev/camera-isp` device file[3]. It subsequently invokes two different `ioctl()` system calls with request types `0xC0086B03` and `0xC0186201` that effectively compromise the driver. As expected, *RootExplorer* deploys the decoy file `/dev/camera-isp` which returns a real file descriptor for `open()`, and success for `ioctl()` (to trick the exploit into believing that it has succeeded). Finally, the exploit performs a `setresuid(0,0,0)` to get root access. At that point, *RootExplorer* successfully finds the root exploit and stops the execution of the app.

In addition to the above two malware samples, VirusTotal also reports three additional malware samples that carry root exploits. We analyzed these cases manually and found that they in fact attempt to download the exploits from a specific URL which is no longer valid. In other words, the exploits are never executed even though the malware may have done it in the past.

9 Limitations and Discussion

Although *RootExplorer* is effective in practice, in theory it has some limitations that would allow attackers to bypass its detection. One obvious limitation is analysis environment evasion (*e.g.,* fingerprinting Android emulator or real phones) which was already discussed in §2. We consider this a general problem for any analysis environment and that this is orthogonal to our research.

There are other limitations specific to our work. First, the signatures that we use are extracted from existing exploits instead of vulnerable code; therefore capable attackers (*e.g.,*, state-sponsored attackers) may be able to find alternative attack paths to exploit the same vulnerability [16]. To address this issue, a different behavior graph thus needs to be learned.

Second, an attacker with knowledge of *RootExplorer* can potentially counter our analysis environment. For instance, without obtaining an actual copy of a device driver (*e.g.,* camera), it is impossible to answer all possible queries from an application. Malware can therefore potentially tell if they are interacting with a real device driver or not. However, we argue that it is also challenging for the malware authors to understand the complete behaviors of a device driver.

Third, since we use syscall-based signatures to model the exploits, *RootExplorer* is also vulnerable to specialized evasion techniques. For example, Ma *et al.* [54] have demonstrated that by splitting the malicious behaviors into pieces that are executed in multiple processes, it is possible to bypass signatures that target a single process. Despite being more difficult, such an attack strategy may also be applicable to certain root exploits and may thus bypass *RootExplorer*'s detection.

We plan to further improve *RootExplorer*'s detection by addressing these problems in the future.

10 Conclusions

In this paper, we tackle the challenging problem of detecting the presence of embedded root exploits in malware. We build a system *RootExplorer*, that learns from commercial-grade root exploits used for benign reasons and backed by large companies such as Baidu and Tencent, and detects such embedded root exploits. Specifically, it carefully analyzes these to determine what environments root exploits expect, and what pre-conditions are to be satisfied in order to trigger them. It uses this information to construct proper analysis environments for malware and can effectively detect the presence of root exploits. Our extensive evaluations shows that it can detect all known malware samples carrying root exploits, and has no false positives. We are also able to detect a root exploit in a malware that seems to have bypassed current vetting procedures, and is available on an Android market.

[3]Note that in this case, the exploit targets a different vulnerability in the same device driver from the example in Section 5.

11 Acknowledgments

This research was partially sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. The work was also partially supported by NSF Award 1617481.

References

[1] Android device inventory. https://www.androiddevice.info/.

[2] Android security bulletin — january 2017, 2018. https://source.android.com/security/bulletin/2017-01-01.html.

[3] 3RD-PARTY ANDROID MARKET. 7723 market, 2017. https://goo.gl/iMi4Bo.

[4] 3RD-PARTY ANDROID MARKET. Android life, 2017. https://goo.gl/hAov2G.

[5] 3RD-PARTY ANDROID MARKET. Eoemarket, 2017. https://goo.gl/FB0ykP.

[6] 3RD-PARTY ANDROID MARKET. Mobomarket, 2017. https://goo.gl/tzpjY7.

[7] AAFER, Y., ZHANG, X., AND DU, W. Harvesting inconsistent security configurations in custom android roms via differential analysis. In *USENIX SECURITY* (2016).

[8] AFONSO, V., BIANCHI, A., FRATANTONIO, Y., DOUPÉ, A., POLINO, M., DE GEUS, P., KRUEGEL, C., AND VIGNA, G. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *Annual Network and Distributed System Security Symposium (NDSS)* (2016).

[9] ANDROGUARD. Androguard, a full python tool to play with android files, 2017. https://goo.gl/edcClw.

[10] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2014), ACM.

[11] AZIM, T., AND NEAMTIU, I. Targeted and depth-first exploration for systematic testing of android apps. In *Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2013), ACM.

[12] BACKES, M., BUGIEL, S., HAMMER, C., SCHRANZ, O., AND VON STYP-REKOWSKY, P. Boxify: Full-fledged app sandboxing for stock android. In *USENIX Security Symposium (Security)* (2015).

[13] BAIDU. Shoujiweishi, 2017. http://shoujiweishi.baidu.com/.

[14] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior-based malware clustering. In *Annual Network and Distributed System Security Symposium (NDSS)* (2009).

[15] BIANCHI, A., FRATANTONIO, Y., KRUEGEL, C., AND VIGNA, G. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)* (2015).

[16] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy (Oakland)* (2006).

[17] BURGUERA, I., ZURUTUZA, U., AND NADJM-TEHRANI, S. Crowdroid: behavior-based malware detection system for android. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)* (2011).

[18] CAO, Y., FRATANTONIO, Y., BIANCHI, A., EGELE, M., KRUEGEL, C., VIGNA, G., AND CHEN, Y. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *Annual Network and Distributed System Security Symposium (NDSS)* (2015).

[19] CHECKPOINT. Quadrooter: New android vulnerabilities in over 900 million devices, 2016. https://goo.gl/GN6ZwW.

[20] CHEN, K. Z., JOHNSON, N. M., D'SILVA, V., DAI, S., MACNAMARA, K., MAGRINO, T. R., WU, E. X., RINARD, M., AND SONG, D. X. Contextual policy enforcement in android applications with permission event graphs. In *Annual Network and Distributed System Security Symposium (NDSS)* (2013).

[21] COPYRIGHT OFFICE, U. Copyright protection and management systems, 2017. https://goo.gl/zpeUtK.

[22] COSTA, M., CASTRO, M., ZHOU, L., ZHANG, L., AND PEINADO, M. Bouncer: Securing software by blocking bad input. In *ACM Symposium on Operating Systems Principles (SOSP)* (2007).

[23] CRANDALL, J. R., SU, Z., WU, S. F., AND CHONG, F. T. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *ACM Conference on Computer and Communications Security (CCS)* (2005).

[24] CUI, W., PEINADO, M., WANG, H. J., AND LOCASTO, M. E. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *IEEE Symposium on Security and Privacy (Oakland)* (2007).

[25] DIMJAŠEVIC, M., ATZENI, S., UGRINA, I., AND RAKAMARIC, Z. Android malware detection based on system calls. *University of Utah, Tech. Rep* (2015).

[26] DINGES, P., AND AGHA, G. Targeted test input generation using symbolic-concrete backward execution. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2014).

[27] DIRTYCOW. Cve-2016-5195, 2017. https://goo.gl/K8cWEK.

[28] ENCK, W., GILBERT, P., HAN, S., TENDULKAR, V., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information-flow tracking system for real-time privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS) 32*, 2 (2014), 5.

[29] FENG, Y., ANAND, S., DILLIG, I., AND AIKEN, A. Apposcopy: Semantics-based detection of android malware through static analysis. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (2014).

[30] FOR INFORMATION SECURITY VULNERABILITY NAMES, S. Cve-2012-6422, 2012. https://goo.gl/R7Icm7.

[31] FOR INFORMATION SECURITY VULNERABILITY NAMES, S. Cve-2014-3153, 2014. https://goo.gl/R7Icm7.

[32] GITHUB. android_run_root_shell (base for godless), 2017. https://goo.gl/VKSWb6.

[33] GITHUB. Cve-2012-6422, 2017. https://github.com/dongmu/vulnerability-poc/tree/master/CVE-2012-6422.

[34] GITHUB. Cve-2014-3153 aka towelroot, 2017. https://github.com/timwr/CVE-2014-3153.

[35] GITHUB. Cve-2015-3636: Poc code for 32 bit android os, 2017. https://github.com/fi01/CVE-2015-3636.

[36] GITHUB. Cve-2015-3636: Poc code for 32 bit android os, 2017. https://github.com/fi01/CVE-2015-3636.

[37] GITHUB. Droidbot, 2017. https://goo.gl/y8ldRA.

[38] GITHUB. exploit for cve-2012-4220 working on zte-open, 2017. https://github.com/poliva/root-zte-open.

[39] GOOGLE. Android and security, 2012. https://goo.gl/mo29A4.

[40] GOOGLE. Dashboards, 2017. https://goo.gl/6BTWw4.

[41] GOOGLE. Security-enhanced linux in android, 2017. https://goo.gl/btJ9xb.

[42] GOOGLE. Verified boot, 2017. https://goo.gl/LiQm9E.

[43] GORDON, M. I., KIM, D., PERKINS, J. H., GILHAM, L., NGUYEN, N., AND RINARD, M. C. Information flow analysis of android applications in droidsafe. In *Annual Network and Distributed System Security Symposium (NDSS)* (2015).

[44] HOFMEYR, S. A., FORREST, S., AND SOMAYAJI, A. Intrusion detection using sequences of system calls. *Journal of computer security 6*, 3 (1998), 151–180.

[45] Hummingbad android malware found in 20 google play store apps, 2016. https://www.bleepingcomputer.com/news/security/hummingbad-android-malware-found-in-20-google-play-store-apps/.

[46] JING, Y., ZHAO, Z., AHN, G.-J., AND HU, H. Morpheus: automatically generating heuristics to detect android emulators. In *Annual Computer Security Applications Conference (ACSAC)* (2014).

[47] JUNOD, P., RINALDINI, J., WEHRLI, J., AND MICHIELIN, J. Obfuscator-llvm–software protection for the masses. In *IEEE/ACM International Workshop on Software Protection (SPRO)* (2015).

[48] KIM, H.-A., AND KARP, B. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium (Security)* (2004).

[49] KOLBITSCH, C., COMPARETTI, P. M., KRUEGEL, C., KIRDA, E., ZHOU, X.-Y., AND WANG, X. Effective and efficient malware detection at the end host. In *USENIX Security Symposium (Security)* (2009).

[50] LI, L., BARTEL, A., BISSYANDÉ, T. F., KLEIN, J., LE TRAON, Y., ARZT, S., RASTHOFER, S., BODDEN, E., OCTEAU, D., AND MCDANIEL, P. Iccta: Detecting inter-component privacy leaks in android apps. In *International Conference on Software Engineering (ICSE)* (2015).

[51] LIANG, Z., AND SEKAR, R. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *ACM Conference on Computer and Communications Security (CCS)* (2005).

[52] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *ACM Conference on Computer and Communications Security (CCS)* (2012).

[53] MA, K.-K., PHANG, K. Y., FOSTER, J. S., AND HICKS, M. Directed symbolic execution. In *International Static Analysis Symposium* (2011).

[54] MA, W., DUAN, P., LIU, S., GU, G., AND LIU, J.-C. Shadow attacks: automatically evading system-call-behavior based malware detection. *Journal in Computer Virology 8*, 1-2 (2012), 1–13.

[55] MICRO, T. Godless mobile malware uses multiple exploits to root devices, 2016. https://goo.gl/qKSCXl.

[56] MUTTI, S., FRATANTONIO, Y., BIANCHI, A., INVERNIZZI, L., CORBETTA, J., KIRAT, D., KRUEGEL, C., AND VIGNA, G. Baredroid: Large-scale analysis of android apps on real devices. In *Annual Computer Security Applications Conference (ACSAC)* (2015).

[57] NETWORKS, P. A. Rootnik android trojan abuses commercial rooting tool and steals private information, 2015. https://goo.gl/epd1IB5.

[58] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy (Oakland)* (2005).

[59] OCTEAU, D., MCDANIEL, P., JHA, S., BARTEL, A., BODDEN, E., KLEIN, J., AND LE TRAON, Y. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *USENIX Security Symposium (Security)* (2013).

[60] PETSAS, T., VOYATZIS, G., ATHANASOPOULOS, E., POLYCHRONAKIS, M., AND IOANNIDIS, S. Rage against the virtual machine: hindering dynamic analysis of android malware. In *European Workshop on System Security (EUROSEC)* (2014).

[61] QIAN, C., LUO, X., SHAO, Y., AND CHAN, A. T. On tracking information flows through jni in android applications. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2014).

[62] RASTOGI, V., CHEN, Y., AND JIANG, X. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security 9*, 1 (2014), 99–108.

[63] SECURELIST. Rooting pokmons in google play store, 2016. https://goo.gl/Ry7AUw.

[64] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated worm fingerprinting. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2004).

[65] SMITH, C. Android statistics, 2016. https://goo.gl/9Pp6I5.

[66] TAM, K., KHAN, S. J., FATTORI, A., AND CAVALLARO, L. Copperdroid: Automatic reconstruction of android malware behaviors. In *Annual Network and Distributed System Security Symposium (NDSS)* (2015).

[67] Upx, 2017. https://goo.gl/6BkD4i.

[68] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: Deterministic rowhammer attacks on mobile platforms. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[69] VIDAS, T., AND CHRISTIN, N. Evading android runtime analysis via sandbox detection. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2014).

[70] Virustotal, 2017. https://goo.gl/Fw7yPC.

[71] WANG, H. J., GUO, C., SIMON, D. R., AND ZUGENMAIER, A. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *ACM SIGCOMM* (2004).

[72] WEI, F., ROY, S., OU, X., ET AL. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[73] WEICHSELBAUM, L., NEUGSCHWANDTNER, M., LINDORFER, M., FRATANTONIO, Y., VAN DER VEEN, V., AND PLATZER, C. Andrubis: Android malware under the magnifying glass. Tech. Rep. TRISECLAB-0414, Vienna University of Technology, 2014.

[74] WONG, M. Y., AND LIE, D. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Annual Network and Distributed System Security Symposium (NDSS)* (2016).

[75] WU, L., GRACE, M., ZHOU, Y., WU, C., AND JIANG, X. The impact of vendor customizations on android security. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[76] XU, L., ZHANG, D., JAYASENA, N., AND CAVAZOS, J. Hadm: Hybrid analysis for detection of malware. In *SAI Intelligent Systems Conference (IntelliSys)* (2016).

[77] XU, W., AND FU, Y. Own your android! yet another universal root. In *USENIX Workshop on Offensive Technologies (WOOT)* (2015).

[78] YAN, L. K., AND YIN, H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security Symposium (Security)* (2012).

[79] YANG, W., XIAO, X., ANDOW, B., LI, S., XIE, T., AND ENCK, W. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *International Conference on Software Engineering (ICSE)* (2015).

[80] YANG, Z., YANG, M., ZHANG, Y., GU, G., NING, P., AND WANG, X. S. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[81] ZHANG, H., SHE, D., AND QIAN, Z. Android root and its providers: A double-edged sword. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[82] ZHANG, H., SHE, D., AND QIAN, Z. Android ion hazard: the curse of customizable memory management system. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[83] ZHANG, M., AND YIN, H. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2014).

[84] ZHENG, C., ZHU, S., DAI, S., GU, G., GONG, X., HAN, X., AND ZOU, W. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)* (2012).

[85] ZHOU, X., LEE, Y., ZHANG, N., NAVEED, M., AND WANG, X. The peril of fragmentation: Security hazards in android device driver customizations. In *IEEE Symposium on Security and Privacy (Oakland)* (2014).

[86] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (Oakland)* (2012).

[87] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Annual Network and Distributed System Security Symposium (NDSS)* (2012).