PREC: Practical Root Exploit Containment for Android Devices

Tsung-Hsuan Ho, Daniel Dean, Xiaohui Gu, William Enck
Department of Computer Science
North Carolina State University
{tho,djdean2}@ncsu.edu, {gu,enck}@cs.ncsu.edu

ABSTRACT

Application markets such as the Google Play Store and the Apple App Store have become the *de facto* method of distributing software to mobile devices. While official markets dedicate significant resources to detecting malware, state-of-the-art malware detection can be easily circumvented using logic bombs or checks for an emulated environment. We present a Practical Root Exploit Containment (PREC) framework that protects users from such conditional malicious behavior. PREC can dynamically identify system calls from high-risk components (e.g., third-party native libraries) and execute those system calls within isolated threads. Hence, PREC can detect and stop root exploits with high accuracy while imposing low interference to benign applications. We have implemented PREC and evaluated our methodology on 140 most popular benign applications and 10 root exploit malicious applications. Our results show that PREC can successfully detect and stop all the tested malware while reducing the false alarm rates by more than one order of magnitude over traditional malware detection algorithms. PREC is light-weight, which makes it practical for runtime on-device root exploit detection and containment.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software (e.g., viruses, worms, Trojan horses)*; D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Android, Host Intrusion Detection, Dynamic Analysis, Malware, Root Exploit

1. INTRODUCTION

Popular application markets (e.g., Apple's App Store, and the Google Play Store) [13, 54] have become a boon for users and developers, but they also provide a distribution point for malware. While markets perform malware analysis (e.g., Bouncer [39]), dynamic analysis environments can be easily detected by malware or avoided using logic bombs and checks for emulation [20, 24, 44]. Thus, it is necessary to provide on-device malware containment that can detect and stop the malware during runtime.

We observe that the dynamic malware analysis performed by application markets provides an opportunity to obtain a *normal behavior model* for an application. In effect, this forces malware authors to *commit* to a behavior during market malware analysis. Therefore, we can use online anomaly detection and malware containment to protect the user from the malware that uses logic bombs or attempts to change execution based on an emulated environment.

The primary challenges in making runtime root exploit containment practical are achieving a low false alarm rate [18] and imposing low interference to benign applications. We address those challenges using two novel techniques. First, we propose a *classified system call monitoring* scheme that can separate system calls based on their origins (e.g., the library that initiates the system call). Thus, we can dynamically identify system calls originated from high-risk components such as third-party native libraries (i.e., native libraries that are not included in the Android system libraries but are downloaded with applications from the app market). Since less than 10% of benign applications include third-party native code [31], the majority of benign applications will have zero false alarms using our scheme. Second, we propose a *delay-based fine-grained containment* mechanism that executes the anomalous system calls using a pool of separate threads, and slows them down exponentially to defeat the attack. The rationale behind our approach is that address space layout randomization (ASLR) in Android [6] forces exploits to repeat the attack sequence many times in order to guess the right stack address. Moreover, most existing root exploits (e.g., Rage Against the Cage) use resource exhaustion attacks (e.g., continuously forking). By slowing down the malicious activity, the exploit becomes unsuccessful and often results in an Application Not Responding (ANR) status, which causes the Android system to kill the malicious application.

In this paper, we present the Practical Root Exploit Containment (PREC) framework to achieve on-device malware containment. We specifically focus on malware that exploits root privilege escalation vulnerabilities. This type of malware represents the highest risk for smartphones, because root access enables the greatest amount of malicious functionality, allows for hiding its existence, and makes the malware difficult to remove [31]. Figure 1 depicts the overall

CODASPY'14, March 3–5, 2014, San Antonio, Texas, USA.

http://dx.doi.org/10.1145/2557547.2557563 .

PREC architecture. PREC operates in two phases: 1) offline learning when a developer submits an app into the market; and 2) online enforcement when the user downloads and installs the app.

When a developer submits an app into the market, the market server (e.g., Google's Bouncer) runs the app within a controlled emulator, performing comprehensive malware detection using a combination of signature detection and dynamic analysis. If the application contains malicious functionality, the market dynamic analysis will detect it and reject the malicious application. However, as mentioned earlier, malware authors often attempt to evade the malware detection system using logic bombs or by not executing malicious code when running in a dynamic analysis environment. This is where PREC provides contribution by forcing the app to commit to a normal behavior.

During dynamic malware analysis, PREC records and labels a system call trace based on our classified monitoring criteria. For this work, PREC labels each system call as originating either from third-party native code or from Java code. We use the third-party native code as the classifying criteria, because it is more risky than Java code or system libraries: 1) all existing malware that exploits root privilege escalation vulnerabilities uses third-party native code; 2) low-level system APIs required by the root exploits are often not available to Java code; and 3) program analysis of third-party native code is significantly more difficult than Java code or system libraries, therefore most malware analysis tools to date ignore third-party native code.

Once system calls are labeled by PREC, it creates a normal behavior model for the app. The normal behavior model is sent to the PREC service that could be hosted within a computing cloud. When choosing a model to represent the normal behavior of an application, we considered several factors such as accuracy, overhead, and robustness to mimicry attacks. After examining several common models such as the hidden Markov model (HMM) and finite state automata (FSA), we developed a new lightweight and robust behavior learning scheme based on the self-organizing map (SOM) technique [35, 23]. SOM is robust to noise in system calls by projecting a high dimensional space (noisy system call sequences) into a two-dimensional map that still captures the principal normal patterns. Moreover, SOM is significantly less computation-intensive than most other learning methods such as HMM.

Ideally, the normal behavior model should be comprehensive in order to avoid false alarms. PREC currently uses a random input fuzz testing tool [11] to create the per-app normal behavior model. Alternatively, app developers can submit an input trace for PREC if a more precise model is desired. Note that PREC is a general behavior learning framework, which can work with any input testing tools or user provided input traces. Our experiments show that using the simple input fuzz testing tool, PREC can already produce high quality behavior models for most of the real Android apps.

The enforcement phase uses the normal behavior model from the PREC service to perform on-device anomaly detection and malware containment. Therefore, PREC dynamically identifies system calls from the third-party native code and performs anomaly detection only on system calls that originate in third-party native code. Monitoring only

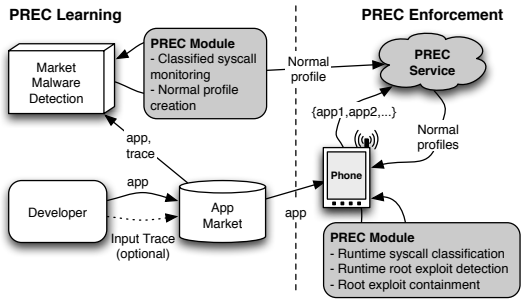


Figure 1: Overview of the PREC architecture. When the developer submits an app to the app market, the market performs extensive malware detection in a controlled emulator. If the app is detected as malware, it is rejected. If not, a normal execution profile is saved and forwarded to the PREC service. When a smartphone user downloads an app, the normal execution profile is retrieved. PREC then monitors operation and contains root exploits on the phone.

third-party native code significantly reduces false alarms of runtime root privilege escalation attack detection.

This paper makes the following contributions:

- We present an architecture for mitigating root exploit malware that hides its existence during dynamic analysis. Our approach forces malware to commit to a normal behavior during market dynamic analysis and malicious attacks are detected and stopped at runtime.
- We describe a runtime, kernel-level, system call origin identification mechanism that allows us to build *fine-grained* behavior models (i.e., third-party native code behaviors v.s. java code behaviors) for higher anomaly detection accuracy and practical malware containment.
- We provide a scalable and robust behavior learning and anomaly detection scheme using the self-organizing map (SOM) learning algorithm [35] that can achieve both high accuracy and low overhead.

We have implemented PREC and evaluated our methodology on 140 most popular benign applications (80 with native code and 60 without native code) covering all different application categories and 10 root exploit malware (4 known root exploit applications from the Malware Genome project [59] and 6 repackaged root exploit applications). Our experiments show that PREC can successfully detect and stop all the tested root exploits. More importantly, PREC achieves practicability by 1) raising 0 false alarm on the benign applications without native code. In contrast, traditional schemes without our classified system call monitoring raise 67-92% per-app false alarms; and 2) reducing the false alarm rate on the benign applications with native code by more than one orders of magnitude over traditional anomaly detection algorithms: from 100% per-app false alarm rate (FSA) and 78% per-app false alarm rate (HMM) to 3.75% per-app false alarm rate (PREC). Since less than 10% apps over the whole market have third-party native code [31], we expect the false alarm rate for PREC will be very low in practice. Our delay-based fine-grained containment scheme can not only defeat all the tested root exploit attacks but also minimize the false alarm impact to the benign applications. Our experiments show that PREC imposes noticeable false alarm impact to

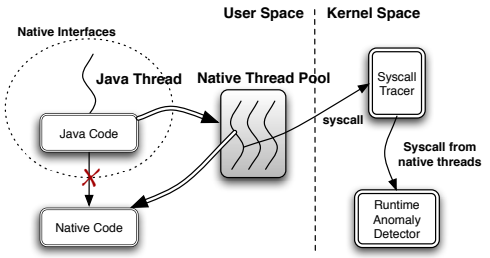


Figure 2: Thread-based system call origin identification. When a third-party native function is called, we dynamically choose a thread from a pool of special "native threads" to execute the function.

only 1 out of 140 tested popular benign applications. PREC is light-weight, which only imposes less than 3% runtime execution overhead on the smartphone device.

2. DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation details of the PREC system. We first describe the system call origin identification scheme. We then describe our on-device root exploit detection schemes. Finally, we present our automatic root exploit containment scheme.

2.1 System Call Origin Identification

PREC performs *classified* system call monitoring by separating the system calls originated from high risk third-party native code from the system calls issued by the less dangerous Java code. However, we cannot simply look at the return address of the code that invokes the system call, because both Java code and third-party native code use system-provided native libraries (e.g., `libc`) to invoke system calls.

Performing user-space stack unwinding from the kernel is one option to understand program components on the call path. However, such backtrace information resides in the user-space and therefore, needs to be well protected. Furthermore, most system libraries do not include debug information (e.g., DWARF [4] or EXIDX [5]) that is needed to unwind the stack.

Another simple approach is to have the Dalvik VM notify the kernel when a thread switches to third-party native code. The kernel can then maintain a flag in the corresponding thread structure in the kernel space. Then, when the native function returns, the Dalvik VM notifies the kernel to clear the flag. This approach makes labeling easy; however it is vulnerable to a confused deputy attack. That is, the kernel cannot determine if it is the Dalvik VM that requested the flag to be unset or a malicious third-party native code.

In light of these limitations, we propose a thread-based approach to identify the system call origins. The basic idea is to maintain a pool of special threads called *native threads* and execute all the third-party native functions using those native threads as shown in Figure 2. Functions executed in these threads are monitored without exception. Malicious code cannot change a native thread to a normal thread. No direct communication is needed between the Dalvik VM and kernel since the pool of native threads are created by PREC at application launch time. Thus, our approach is not vulnerable to the confused deputy attack. Furthermore, the code logic for determining the switching between native threads and java threads is in Dalvik VM which Android ensures is read-only to the application process [1]. Thus, the malicious attacker cannot bypass the switching from java threads to native threads in order to execute the third-party native code.

We build our system call tracer as a Linux kernel module on top of kprobes. Kprobes are a set of Linux interfaces that allow us to implant probes and register corresponding handlers. Compared to user space system call tracers (e.g., ptrace [7]) that can introduce over 20% overhead due to frequent context switches, our kernel tracer only incurs less than 2% overhead. PREC could also use other kernel space system call tracing tools such as SystemTap [10], DTrace [3], or Linux Trace Toolkit - next generation (LTTng) [8] that are orthogonal to our approach.

Our system call origin identification scheme leverages the natural boundary between Java code and native code. Android allows Java components to access native binaries (including both libraries and executables) in three different ways. First, Java components can use the JNI Bridge to call native functions. The JNI Bridge is the aggregation point that maps Java abstract native function names (i.e., `static native function` appeared in Java code) to real native function addresses. Second, when the Java code requires a native library to be loaded, `System.loadLibrary()` will load the native library to the memory and then call the `JNI_OnLoad()` callback in the library. Since `JNI_OnLoad()` is defined by the developer of the native library, it can be used by an adversary to execute native code. Lastly, Java allows applications to use `System.Runtime.exec()` to execute native executables. This function is the best place for attackers to apply root privilege escalation attacks because most exploits are released as native executables. For the rest of this paper, we use *native interfaces* to represent different ways to invoke third-party native functions.

When a Java thread needs to invoke a third-party native function through one of the aforementioned native interfaces, PREC is triggered to suspend the Java thread and use a native thread to execute the native function instead. One brute force implementation is to create a new native thread each time the native function is invoked. However, this simple implementation suffers from high performance overhead when the application frequently invokes native functions. Instead, PREC creates a pool of native threads at application launch. When a Java thread needs to execute a third-party native function, we suspend the Java thread and dynamically select an idle native thread to execute the native function. The native function sometimes calls back to the Java code (e.g., `NewStringUTF()`, which is a function that creates a Java string inside the Java heap). Under those circumstances, we continue use the native thread to execute the Java function because it might be a potential attack to Java components. When the native function exits, we resume the Java thread and recycle the native thread.

Our thread-based system call origin identification scheme has several advantages over other alternatives. First, the kernel tracer can easily identify the system call origins (i.e., from Java or native components) by checking whether the thread belongs to the native thread pool. Second, the thread-based approach allows us to isolate a small portion of the application rather than kill the whole application. This allows us to reduce the disturbance to the user by minimizing the containment scope. We will describe our containment

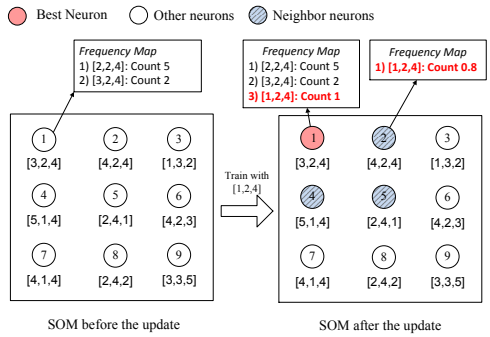


Figure 3: SOM update example using the input vector [1,2,4].

scheme in Section 2.3. Third, PREC can easily incorporate other execution sandboxing mechanisms (e.g., software fault isolation [57]) to provide additional security isolations between Java code and malicious native code.

2.2 On-Device Root Exploit Detection

After we extract the system calls from the high-risk native code, we need to build a normal behavior model for the app before it is released to the market. The behavior model is then transferred to the smartphone device for runtime root exploit detection.

Normal app behavior learning. We capture the normal behavior of each application during the market dynamic malware analysis. As mentioned in the Introduction, we develop a new lightweight and robust behavior learning scheme based on the self-organizing map (SOM) technique [35]. Our previous work [23] has used SOM to train a system behavior model using system-level metrics such as CPU usage and memory consumption. To the best of our knowledge, this work makes the first step in applying SOM to system call sequences.

SOM is a type of artificial neural network that is trained using unsupervised learning to map the input space of the training data into a low dimensional (usually two dimensions) map space. Each map consists of $n \times m$ nodes called *neurons* arranged in a grid, illustrated by Figure 3. Each neuron is associated with a *weight vector* that has the same length as the input vector. In our case, both input vectors and weight vectors are sequences of system call identifiers (ids) of length k (i.e., k-grams). Both n, m, and k are configurable parameters that can be dynamically set during map creation. At map creation time, each weight vector element is initialized randomly to be a value i such that $1 \leq i \leq S$, S equals to the largest system call id. In order to handle applications with different behaviors, PREC builds a SOM for each individual application and only uses the system calls originated by high-risk third-party native code to train the SOM.

The traditional SOM learning algorithm updates weight vectors continuously. However we cannot use this method directly, since two system calls with similar ids do not necessarily have similar actions. For example, system call id 12 (`sys_chdir`) is completely different than system call id 13 (`sys_time`). To address these issues, we have made two modifications to the traditional SOM learning algorithm. First, we use the string edit distance instead of Euclidean or Manhattan distance as a measure of similarity when mapping input vectors to neurons. This is because graph edit distance only considers if two items are exactly the same in the weight vector. Second, to address the continuous update problem, we have developed a frequency-based weight vector update scheme, which we describe next.

Each SOM model training occurs in three iterative steps, illustrated by Figure 3. First, we form an input vector of length k by reading k system calls from the training data. Second, we examine the string edit distance from that input vector to the weight vectors of all neurons in the map. Whichever neuron has the smallest distance is selected as the winning neuron to be trained. We break ties using Euclidean distance. Third, we add 1 to the count for the input vector in the *frequency map* of the winner neuron. At this point we also update the frequency maps of all neighbor neurons. In this example, we define our neighborhood to be the neurons in a radius of $r = 1$. The count value added to the neighbor neuron is reduced based on a neighborhood function (e.g., Gaussian function) which depends on the grid distance from the winning neuron to the neighbor neuron. For example, in Figure 3, the input vector $[1, 2, 4]$ is added into the frequency map of the winning neuron 1 with a count 1 and is also added into the frequency map of the neighbor neuron 2 with a reduced count 0.8.

The frequency map keeps track of how many times each particular system call sequence has been mapped to that neuron. For example, in Figure 3, the frequency map of neuron 1 shows that the sequence $[2, 2, 4]$ is mapped to the neuron 1 five times, the sequence $[3, 2, 4]$ is mapped to neuron 1 two times, and the sequence $[1, 2, 4]$ is mapped to neuron 1 just once. We repeat the above three steps for all the system call sequences recorded in the training data. After training is complete, we use the sequence with the highest count in the frequency map to denote the weight vector of the neuron. We sum the count values of all the sequences in the frequency map to denote the frequency count value for this neuron.

Use of system call arguments. System call arguments provide finer-grained information to system call anomaly detections. In PREC, we selected two types of arguments to help detect root exploits: file paths and socket arguments. We divide each file and socket related system call into multiple subgroups based on the arguments it contains. Specifically, we classify file paths into two types: application accessible directories and system directories. We divide socket system calls into three different groups based on its protocol type: 1) the socket call that connects to a remote server on the network, 2) a local server on the device, and 3) a kernel component with the `NETLINK` socket. Each file or socket system call is assigned with different identifiers based on the argument type. For example, the system call `open` is assigned with an identifier 5 for accessing its home directory or SD card partition and a different identifier (e.g., 300) for accessing the system directories.

Some system calls (e.g., `symlink`, `rename`) include two file paths in their arguments. If the two file paths belong to the same type, we can assign the system call identifier in a similar way as single file path ones. However, if the two file paths belong to different types, we assign a unique identifier to the system call. The intuition behind our approach is that we observe that benign applications do not simultaneously access files in the application home directory and the system directory. For example, benign applications do not move files from its home directory to system partitions

and vice versa. In contrast, we observe that most malicious applications try to access home directories and system directories at the same time (e.g., symlink a system file to a local directory).

Runtime root exploit detection. When a user purchases an app from the market, its normal behavior model represented by SOM is downloaded to the user's smartphone. After the application starts, PREC performs runtime system call origin identification to form the sequences of system calls originated by the third-party native code. We then match the system call sequences against the SOM model. If a root exploit begins to execute, PREC identifies system call sequences that are mapped to rarely trained neurons. Thus, if we map the collected system call sequence to a neuron whose frequency count is less than a pre-defined threshold (e.g., 0 represents never trained), the current sequence is considered to be malicious. The threshold allows users to control the tradeoff between malware detection rate and false alarm rate. The map size and the sequence length are other configuration parameters that might contribute to the malware detection accuracy tradeoff. We will quantify such tradeoffs in the experimental evaluation section.

2.3 Automatic Root Exploit Containment

When a root exploit is detected, PREC automatically responds to the alarm by containing the malicious execution. A brute force response to the malware alarm would be killing the entire application to protect the device from the root compromise. However, this brute force approach might cause a lot of undesired disturbances to the user, especially when the anomaly detector raises a false alarm. To address the challenge, PREC provides fine-grained containment by stopping or slowing down the malicious activities only instead of the whole application.

As mentioned in Section 2.1, PREC executes system calls from the third-party native code within the special native threads. When a malicious system call sequence is detected, PREC sends a predefined signal to the malicious native thread to terminate the thread. To process the signal, we also insert a signal handler inside the native thread before the native function is called. In our current prototype implementation, we use `SIGSYS` (signal 31) to trigger the native thread termination. We confirm that `SIGSYS` is not used by any other Android system components. Furthermore, PREC disallows applications from sending or registering handlers for `SIGSYS`.

Although killing native threads can effectively stop the attack, it might still break the normal application execution when the anomaly detector raises a false alarm. Thus, PREC provides a second containment option that is less intrusive: slowing down the malicious native thread by inserting a delay during its execution. Our experiments show that most root exploits become ineffective after we slow down the malicious native thread to a certain point. The delay-based approach can handle the false alarms more gracefully since the benign application will not suffer from crashing or termination due to transient false alarms.

To insert delay into the malicious thread, we force the kernel to call our sleep function before each system call is dispatched to the corresponding handler. After the anomaly detection module raises an alarm, it sets a delay value in the `task_struct` of the malicious native thread. Note that the `task_struct` is ready-only to the user process. Thus, PREC pauses the native thread based on the delay specified by PREC. The delay time is applied to all subsequent system calls in the thread, and exponentially increases for each new alarm in order to stop the malicious activities in a timely way. Our prototype starts at 1 ms and doubles per alarm. For each normal system call sequence, we exponentially decrease (halves) the delay. There are other possible policies for increasing or decreasing the delay, which can be used as tuning knobs for controlling the false alarm sensitivity. For example, we also tested a linear decrease policy, but found exponential decrease can tolerate more false alarms.

3. EXPERIMENTAL EVALUATION

We implement PREC and evaluate our approach using real world root exploits and applications. We evaluate PREC in terms of detection accuracy, malware containment effectiveness, and overhead.

3.1 Evaluation Methodology

Benign application selection: We first test PREC with a variety of popular benign apps to evaluate the false alarm rate of PREC. We select our benign apps as follows. We downloaded top 10 popular free apps from all different application categories (Android Market includes 34 application categories). We then test those applications from the most popular ones to less popular ones and check whether we can run them successfully on the emulator and our Samsung Galaxy Nexus device. We find 80 popular apps include third-party native code and can be correctly executed on our test devices. The majority of them are games and multimedia applications. We also test 60 popular apps without any third-party native code. We use more benign apps with third-party native code than without third-party native code in order to estimate the worst-case false alarm rate of PREC since PREC will not raise any false alarm for benign apps without any third-party native code. In contrast, other alternative schemes without our classified monitoring techniques will still raise false alarms on those benign apps without third-party native code. We evaluated all the benign apps using a Samsung Galaxy Nexus device with Android 4.2, which is equipped with 1.2 GHz Dual-Core cortex A9 processor, and 1GB RAM.

Malware selection: To evaluate the root exploit containment capability of PREC, we extensively studied all the existing real root exploits. Table 1 shows the 10 malicious applications used in our experiments that covers four real root exploits. We first used four real malware samples reported by the Malware Genome project [59]. To evaluate PREC under more challenging cases, we repackage existing root privilege escalation attacks into a popular application (AngryBirds), which contains a lot of native code.

We first studied all the six root exploit malware families (DroidDream, DroidKungFu1, DroidKungFu2, Ginger Master, BaseBridge, DroidKungFuSapp) reported by the Malware Genome Project [59]. Our experiments covered the first four malware families. The BaseBridge malware only attacks Sony and Motorola devices, which cannot be triggered on our Nexus phones. The DroidKungFuSapp performs attacks by connecting to a remote command and control server. However, the server was inaccessible at the time of our testing, which did not allow us to trigger the root exploit.

Table 1: Malware samples tested in the experiments. The first 4 malware samples are existing malware and the last 6 malware samples are repackaged AngryBirds applications with existing root exploits.

Malware Sample	Application Package	Root Exploits	Description
DroidDream	com.beauty.leg	Exploid, **RATC**	Plaintext root exploits which are triggered once the infected application has been launched
DroidKungFu1	com.sansec	Exploid, **RATC**	Encrypt Exploid and RATC root attacks and is triggered when a specific event is received at a predefined time condition.
DroidKungFu2	com.allen.txtdbwshs	Exploid, **RATC**	Encrypt the RATC root attack and is triggered when a specific event is received at a predefined time condition.
GingerMaster	com.igamepower.appmaster	**GingerBreak**	Hides shell code suffix and is triggered during the next reboot.
RATC-1	AngryBirds	**RATC**	Attack is triggered when the application receives `BOOT_COMPLETED` broadcast intent.
RATC-2	AngryBirds	**RATC**	Attack is triggered ten seconds after the application is launched.
ZimperLich-1	AngryBirds	**ZimperLich**	Attack is triggered when the application receives `BOOT_COMPLETED` broadcast intent.
ZimperLich-2	AngryBirds	**ZimperLich**	Attack is triggered ten seconds after the application is launched.
GingerBreak-1	AngryBirds	**GingerBreak**	Attack is triggered when the application receives `BOOT_COMPLETED` broadcast intent.
GingerBreak-2	AngryBirds	**GingerBreak**	Attack is triggered ten seconds after the application is launched.

The RiskRanker project [31] and the X-ray project [12] reported 9 root exploits in total. Our experiments covered four of them (Exploid, RATC, GingerBreak, ZimperLich). We did not cover the other five root exploits for the following reasons. Three reported root exploits (Ashmem, zergRush, Mempodroid) are not found in real Android applications. Ashmem uses a vulnerability that Android failed to protect *Android Share Memory* so unprivileged process can change the value of `ro.secure` arbitrarily. This variable is used by the Android Debug Bridge Daemon (ADBD) to determine whether developer can login as *root*. However, attackers cannot embed this exploit into applications because Android applications cannot access ADBD. Similarly, *zergRush* requires several information in ADBD and *Mempodroid* executes `run-as` inside the Android Debug Bridge shell. Therefore, it is infeasible for attackers to use those exploits in applications. The remaining two root exploits (*Asroot*, which is named *Wunderbar* in X-ray, and *Levitator*) are not tested due to lack of software or hardware: *Asroot* targets on Linux kernel version prior 2.6.30-4, and the earliest available version that we can use for Nexus One device is 2.6.32. *Levitator* targets PowerVR driver and our Nexus One device uses Adreno 200 GPU. However, we also studied the source code of *Asroot* and *Levitator* and confirmed that PREC can detect those two root exploits if they are triggered. The reasons are that they either use some system calls that should never be used by normal applications (e.g., syscall 187 in *Asroot*) or need to repeatedly execute certain system calls (similar to GingerBreak) to achieve success.

We tested all the root exploit malware on a Google Nexus One device with Android 2.2 with 1GHz single core cortex A8 processor and 512MB RAM. Although the latest root exploit in our data set targets Android 2.3, root privilege escalation attacks are an increasing concern in Android. For example, Google introduced SELinux in Android 4.3 to mitigate the damage of root escalation attacks [9]. PREC provides a complementary first-line defense to detect and contain the root escalation attacks.

Model learning data collection in emulator: All the application behavior model learning data were collected on the Android emulator enhanced with our classified system call monitoring scheme. We used the Android Monkey [11] tool to generate random inputs to simulate user behaviors. We chose Monkey in this work because it is the best publicly available fuzz input generation tool we could find at the time of writing. Previous work [52] also shows that Monkey can provide similar coverage as manual collection given sufficient rounds of testing. We note that using Monkey input generation is a limitation in our current implementation, which will be discussed in detail in Section 4. However, our experiments show that PREC can achieve high accuracy even by using such a simple tool. We expect PREC can achieve even more accurate malware detection given a more powerful input generation tool or using developer provided input traces. Although previous work [58, 56, 49, 41] proposed to automate the trace collection process by analyzing decompiled Java source code and standard Android user interface (UI) components, those approaches cannot be applied to PREC for two main reasons. First, PREC focuses on third-party native code which is very difficult, if not totally impossible, to decompile. Second, most applications that contain native code do not use standard UI components. Rather, they often draw UI components themselves.

Each application learning data collection lasted 10 minutes. For benign applications, trace collection was performed on a modified Android 4.2 emulator (API level 17). We collected traces for malicious applications on a modified Android 2.2 emulator (API level 8) because they require Android 2.2 to trigger the exploits. Note there is no root exploit triggered in the training data collection phase since we assume that malware try to hide themselves in the dynamic analysis environment using logic bombs or detecting emulation. If the root exploit is triggered, the malicious activities will be detected by the market malware analysis and the application will be rejected.

On-device real application testing data collection: To evaluate the on-device benign application false alarm rates and malware detection accuracy of PREC, we use real users to run all the 140 benign applications on our Samsung Galaxy Nexus device with Android 4.2 for collecting

realistic user behaviors. For each app, the user is asked to play the app for about three minutes. Although we could also use the same dynamic testing tool to collect the testing data automatically, we chose not do so to avoid producing biased results using the same tool for both learning and testing. For those 10 malicious applications listed in Table 1, we run them on a Google Nexus One device with Android 2.2 and make sure those root exploits are triggered during our testing phase.

Alternative algorithms for comparison: In addition to PREC, we also implement a set of different anomaly detection schemes for comparison: 1) *SOM (full)* that applies the SOM learning algorithm over *all* system calls to create normal application behavior models; 2) *HMM (native)* that applies the hidden Markov model [53] over the system calls from the third-party native code only, which learn normal system call sequence transition probabilities and raises an alarm if the observed tradition probability is below a threshold; 3) *HMM (full)* that uses the hidden Markov model over all system calls; 4) *FSA (native)* [42] that uses the finite state automaton over the system calls from the third-party native code only, which learns normal system call sequence patterns and raises an alarm if the observed system call sequence transition probability is below a pre-defined threshold; and 5) *FSA (full)* that uses a finite state automaton over all system calls. Note that we only compare PREC with common *unsupervised* learning methods since supervised learning methods (e.g., support vector machine [32]) cannot be applied to PREC as they require malware data during the learning phase and cannot detect unknown malware.

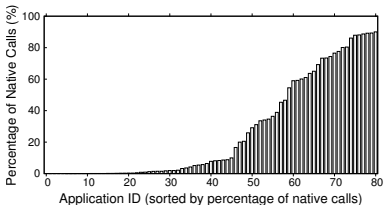
Evaluation metrics: We evaluate the malware detection accuracy using the standard *receiver operating characteristic* (ROC) curves. ROC curves can effectively show the tradeoff between the true positive rate (A_T) and the false positive rate (A_F) for an anomaly detection model. We use standard *true positive rate* A_T and *false positive rate* A_F metrics, as shown in Equation 1. N_{tp}, N_{fn}, N_{fp}, and N_{tn} denote the true positive number, false negative number, false positive number, and true negative number, respectively.

$$A_T = \frac{N_{tp}}{N_{tp} + N_{fn}}, \quad A_F = \frac{N_{fp}}{N_{fp} + N_{tn}} \tag{1}$$

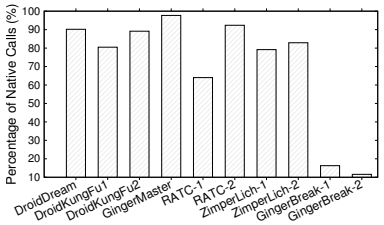
A false positive means that our anomaly detection system raises an alarm for a benign application. A false negative means that we fail to raise any alarm for a malware sample. In our results, we report both per-sequence (i.e., system call sequence) and per-app true positive rates and false positive rates.

3.2 Results and Analysis

Runtime classified system call monitoring: We first evaluate the effectiveness of our runtime classified system call monitoring module that serves as the foundation for PREC. Figures 4(a) shows the percentage of the system calls originated from the third-party native code for the 80 benign apps that include third-party native code. Although all those 80 apps contain native code, we observe that over 50% of the apps execute less than 10% third-party native code. Thus, PREC can still filter out a large number of system calls for those benign applications with third-party native code during model creation and malware detection.

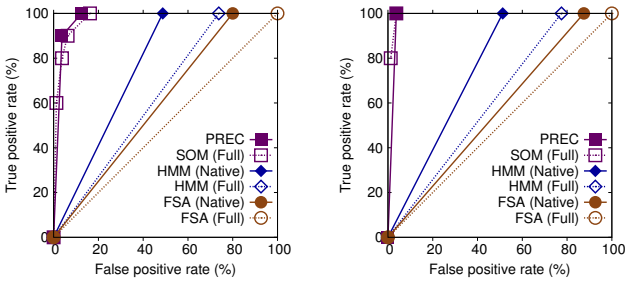


(a) 80 benign apps with native code



(b) 10 Malicious apps

Figure 4: Percentage of system calls originated from native code for 80 apps with third-party native code and 10 malicious apps.

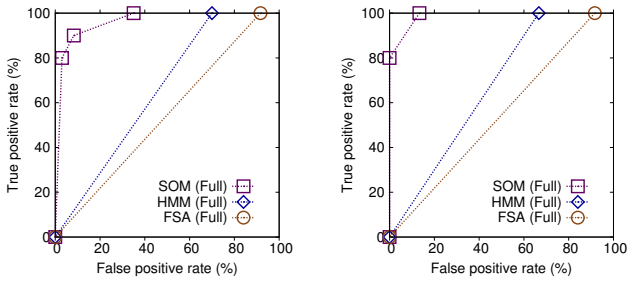


(a) Without arguments (b) With arguments

Figure 5: Per-app malware detection accuracy comparison results for 80 apps that include third-party native code.

We also observe that PREC never misclassifies a system call from Java as a system call from third-party native code. Thus, PREC will not raise any false alarm for those benign applications that do not include any third-party native code. Figure 4(b) shows that the classified monitoring results for the 10 malware samples used in our experiments. We can see most malware applications contain a large portion of system calls from the third-party native code. This also validates our hypothesis: malware exploits root privilege escalation vulnerabilities using third-party native code. Thus, our classified monitoring scheme will not reduce the root exploit detection capability.

Runtime on-device detection accuracy: We now evaluate the runtime on-device detection accuracy of the PREC system. Figure 5 shows the per-app true positive and false positive rate using different anomaly detection algorithms for the 80 benign apps that include third-party native code. Figure 5(a) shows the results without considering system call arguments while Figure 5(b) shows the results of including system call arguments. We adjust different parameters in each anomaly detection algorithm to obtain the ROC curves. For SOM algorithms, we adjusted the map size, the length of the system call sequences, the anomalous frequency thresh-



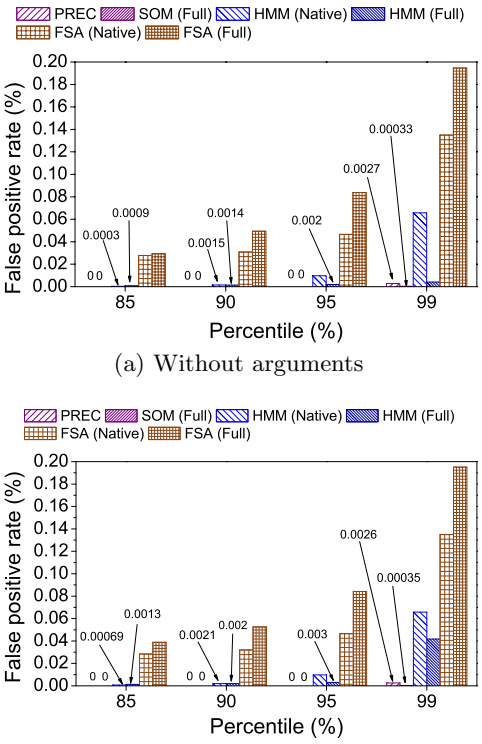
(a) Without arguments (b) With arguments

Figure 6: Per-app malware detection accuracy comparison results for 60 apps that do not have any native code.

old, and the neighborhood area size. For HMM and FSA algorithms, we adjusted the number of states, the system call sequence length, and the anomalous probability threshold. Each point on the ROC curve represents one accuracy result under a certain configuration setting and we use the configuration for all the 80 apps. If two configurations produce the same true positive rate but different false positive rates, we only plot the point with the smaller false positive rate to show the best accuracy result of each scheme. The results show that all algorithms can easily achieve 100% true positive rate but their false positive rates vary a lot. HMM and FSA can achieve 49% and 80% false positive rate at their best, respectively. In contrast, PREC can significantly reduce the false positive rate to 3.75%. This validates our choice of SOM since it is much more robust to noise in system calls than HMM and FSA by projecting the original noisy input space (noisy system call sequences) into a two-dimensional map without losing principal patterns.

We believe that the user perceived false positive rate of PREC will be even lower since most popular benign apps do not include third-party native code and PREC will not raise any false alarm on them with the help of classified system call monitoring. However, without performing classified system call monitoring, any anomaly detection algorithm might still raise false alarms on those apps without third-party native code. Figure 6 shows the anomaly detection accuracy results of different algorithms without using the classified system call monitoring scheme for 60 benign apps without third-party native code. We observe that SOM (full), HMM (full), and FSA (full) raise 13%, 67%, and 92% per-app false alarms at their best under 100% true positive rate. This validates our hypothesis that classified monitoring can greatly reduce the false alarms in practice during runtime root exploit detection.

We further compare different anomaly detection algorithms at fine granularity by measuring per-sequence false positive rates. Figure 7 shows the per-sequence false positive rates at 85-99 percentile achieved by different schemes for the 80 benign apps that include third-party native code. For fair comparison, we pick the configuration for each algorithm that yields the best per-app anomaly detection accuracy result for the algorithm. We observe that SOM algorithms can reduce the per-sequence false positive rates by more than one order of magnitude compared to HMM and FSA. Figure 8 shows the per-sequence false positive rate comparison for the benign apps without any third-party native code. We also



(a) Without arguments

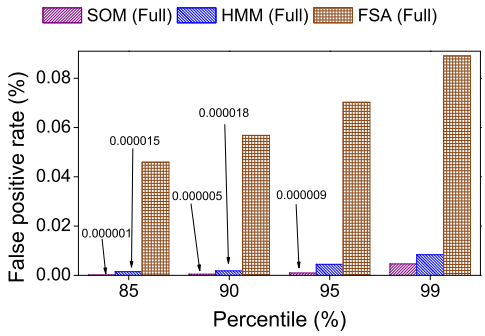
(b) With arguments

Figure 7: Per-sequence false positive rate comparison for 80 apps that include third-party native code.

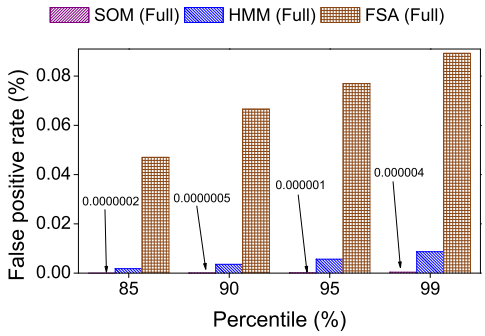
observe that SOM can significantly reduce the false positive rate by orders of magnitude.

Malware containment results: We now evaluate the malware containment effectiveness of PREC. We trigger each malicious app on the smartphone and run the PREC system on the phone to see whether it can stop the root exploit attack. Table 2 summarizes our malware containment results. As mentioned in Section 2.3, PREC provides two different containment mechanisms: 1) *termination-based containment* that stops the root exploit attack by killing the malicious native threads and 2) *delay-based containment* that stops the root exploit attack by inserting exponentially increasing delays in anomalous system calls. The results show that our anomaly detection can successfully detect and stop all the root exploit attacks before they succeed. We measure the alarm lead time as the time elapsed between the when root exploit is detected and when the root exploit is successful if no containment scheme is taken. For the repackaged malicious applications (RATC-1, RATC-2, ZimperLich-1, ZimperLich-2, GingerBreak-1, GingerBreak-2), we can terminate the malicious native threads only and continue to run the AngryBirds application normally.

We further analyze which system call sequences first cause our anomaly detector to raise alarms. For the GingerBreak and both repackaged RATC malware samples, PREC detects the abnormal sequence `[execve, execve, execve, execve, close, getpid, sigaction, sigaction, sigaction]`. This is consistent with the behaviors of those root exploits which first copy exploit files to a given directory and execute `chmod` executable to change permission to `0755` for later execution. Because different devices place `chmod` in different



(a) Without arguments



(b) With arguments

Figure 8: Per-sequence false positive rate comparison for 60 apps that do not include any third-party native code.

directories, the root exploit must try several locations to find the right directory. We omit the detailed malicious system call sequence results for other malware samples due to space limitation. In summary, our sequence analysis results show that PREC can accurately catch the malicious behaviors of those malware samples.

False alarm impact results: We now evaluate the false alarm impact of our system using different containment schemes. As shown in Figure 5 and 6, PREC only raises false alarms in 3 (out of 140 tested) benign apps (Forthblue.pool, TalkingTom2Free, CamScanner). We first tried the *termination-based containment* over those three benign applications. We found that those applications crashed even if we only killed the malicious native threads. We then tested our *delay-based containment scheme* over these three apps. If we only insert delays in malicious native threads, we observed that our containment scheme incurs negligible impact (0.1-0.25 second total delay during 3 minutes run) to the two benign applications, TalkingTom2Free and CamScanner. Forthblue.pool hangs after the delay-based containment is triggered. To summarize, PREC only incurs significant false alarm impact to 1 out of 140 benign popular apps tested in our experiments.

PREC overhead results: We first evaluate our anomaly detection overhead. Table 3 shows the per-app model training time and per-sequence anomaly detection time comparison among different algorithms. We can see both SOM and FSA algorithms are light-weight. However, FSA tends to raise a large number of false alarms, which makes it impractical for runtime malware detection. HMM is sensitive

Table 2: Malware detection and containment results.

Malware Samples	Alarm lead time	Termination-based containment	Delay-based containment
DroidDream	20.7 sec	success	success
DroidKungFu1	16.1 sec	success	success
DroidKungFu2	96.5 sec	success	success
GingerMaster	318.3 sec	success	success
RATC-1	26.6 sec	success	success
RATC-2	17.1 sec	success	success
ZimperLich-1	14.1 sec	success	success
ZimperLich-2	20.9 sec	success	success
GingerBreak-1	35 sec	success	success
GingerBreak-2	34.6 sec	success	success

Table 3: Anomaly detection model training and runtime detection time comparison. For HMM, "$S = i$" means the number of states is configured to be i in HMM. "$S = max$" means the number of states equal to the number of distinctive system calls in the trace. The average per-app system call sequence number is 244K under all system call monitoring and 106K under native thread system call monitoring.

Scheme	Per-app training time	Per-sequence detection time
PREC	39.7 ± 59.3s	0.07 ± 0.03ms
SOM (full)	131.3±88.6s	0.12 ± 0.00001ms
HMM (S= 10, native)	11.8 ±15.9s	1.1 ± 2.7ms
HMM (S= 10, full)	32.3 ±22s	0.2 ± 0.3ms
HMM (S= 20, native)	72.5±107.7s	8.8 ± 20.4ms
HMM (S= 20, full)	140.8±121.8s	1.8 ± 2.7ms
HMM (S = max, native)	1040 ± 2123s	7.7 ± 13ms
HMM (S = max, full)	2449 ± 1834.2s	105.9 ± 143.2ms
FSA (native)	0.6 ±1s	0.05 ± 0.26ms
FSA (full)	1.1 ±0.7s	0.01 ±0ms

to the number of states configured in the model. As we increase the number of states to its maximum value (i.e., the number of distinctive system calls used in the training trace), the overhead of HMM becomes too large to be practical. Although we see increased detection accuracy as we increase the number of states, the best case of HMM is still much worse than SOM, as shown in our detection accuracy results. To quantify the runtime overhead of PREC, we run PREC on a Galaxy Nexus phone with Andriod 4.2 using Antutu benchmarks [2]. Figure 9 shows the benchmark performance results. We observe that our classified monitoring scheme imposes less than 1% overhead and the SOM anomaly detection algorithm imposes up to 2% overhead. Overall, PREC is light-weight, which makes it practical for smartphone devices.

4. LIMITATION DISCUSSION

Our current trace collection prototype implementation has two limitations. First, simple fuzz-testing tools such as the Monkey tool are not guaranteed to produce sufficient behavioral coverage. However, our experiments show that this simple tool works fine for the apps tested in our experiments. One method of overcoming this limitation is to perform the trace collection with a more fine-tuned automation process such as event triggering and intelligent execution [49]. Developers can also provide application input traces, which

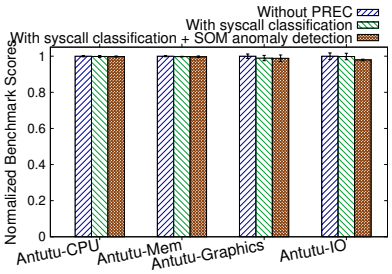


Figure 9: PREC runtime performance overhead under different benchmark apps on Galaxy Nexus running Andriod 4.2.

allow us to collect training data based on the developer-provided input trace for sufficient coverage. Second, a few applications use different libraries (e.g., OpenGL library) when the application runs in the real device and in the emulator due to the hardware difference between the real device and the emulator host. To eliminate the impact of this library difference to our results, we exclude those applications that use OpenGL library in our experiments. One solution to this limitation is to collect the training data for those applications using special libraries on real devices offline to eliminate the issue of platform difference [47].

PREC currently only monitors system calls from the third-party native code. Thus, it is possible to have false negatives if the attacker performs root exploits from the Java code. While it is theoretically possible for the root exploit to originate from Java code, root exploits require access to low-level APIs that are difficult to execute purely in Java. Thus, our approach raises the bar for attackers. However, we note this as a limitation and topic of future research.

Any system call based anomaly detection schemes is susceptible to mimicry attacks. SOM includes randomization as part of its learning, which makes it potentially more robust to mimicry attacks. We can initialize the map with different random seeds for different users. The resultant normal profiles will look similar, but not exactly the same. Therefore, our approach makes it harder for attackers to infer the exact normal model for each user and succeed in hiding the malicious system call sequences in normal behaviors on all the attacked devices.

5. RELATED WORK

Forrest et al. [26] first proposed the system call malware detection schemes by building a database of *normal* system call sequences. Warrender et al. [53] extended this idea by using hidden Markov models (HMMs) to model sequences of normal system calls. Other researchers [16, 34, 22] adapted artificial neural network to perform intrusion detection. Kruegel et al. [37] proposed to use system call arguments to improve the performance of host-based intrusion detection. Maggi et al. [40] proposed to cluster similar system calls or similar system call arguments to further improve the accuracy. Previous work [30, 38] also used SOM for network intrusion detection by clustering system call arguments such as user name, connection type, and connection time. Gao et al. [28, 27] perform real-time anomaly detection based on differences between execution graphs and the replica graphs constructed using system call traces and runtime information (e.g., return addresses).Traditional system call based intrusion detection approaches have to collect *all* system calls made by the target process, as there is no clear boundary to reduce the collection scope. This increases both noise and design complexity of intrusion detection. In contrast, PREC leverages the natural component boundary in the Android system to perform *classified* system call monitoring. As a result, PREC can achieve more accurate and practical malware detection.

Crowdroid [17] collects system calls on smartphones and sends system call statistics to a centralized server. Based on the theory of crowdsourcing, symptoms that are shared by a small number of devices are treated as abnormal. Similarly, Paranoid Android [48] runs a daemon on the phone to collect behaviors and a virtual replica of the phone in the cloud. The daemon transmits collected behaviors to the cloud and the virtual phone replays the actions happening on the smartphone based on the collected behaviors. Both Crowdroid and Paranoid Android incur 15-30% overhead to smartphone devices. In contrast, PREC imposes less than 3% overhead to the smartphone device, which makes it practical for runtime smartphone malware containment. SmartSiren [21] gathers and reports communication information to a proxy for anomaly detection. Besides the runtime overhead, propagating sensitive communication data to a public server might be a privacy concern for users. In contrast, PREC does not require any smartphone data to be sent to remote servers.

Recent work has explored using specific subsets of system calls for smartphone security. Isohara et al. [33] monitor a *pre-defined* subset of system calls such as `open` on smartphones. pBMDS [55] hooks input-event related functions (e.g., `sys_read()` for keyboard events, specific drivers for touch events) to collect system calls related to user interaction behaviors (e.g., GUI events). It then performs malware detection using HMMs. In contrast, PREC selects system calls based on their origins rather than pre-defined system call types. To the best of our knowledge, PREC makes the first step in classifying system calls based on their origins to significantly reduce the false alarms during runtime malware detection.

Moser et al. [43] monitor system calls executed when a program tries to terminate, with the intention of understanding how malware evades detection in virtualized test environments. Bayer et al. [15] create malicious application behavioral profiles by combining system call traces with system call dependency and operation information. Kolbitsch et al. [36] generate hard to obfuscate models that track the dependencies of system calls in known malware, which they then can use to detect malware. CloudAV [45] intercepts every `open` system call and extracts the target file. It compares this signature with signatures of abnormal files maintained in the cloud to detect the access of malicious files. DroidRanger [60] utilizes a signature-based algorithm to detect known malware from markets and monitors sensitive API accesses to discover zero-day malware. In contrast, PREC does not train on malware and can detect zero-day malware that hides itself during market malware analysis. Several researchers have also applied machine learning to statically detect malicious applications based on their permissions [50, 46, 19]; however, the root exploit malware addressed by PREC does not require permissions.

Previous work has been done to automatically respond to malicious attacks on networked hosts. For example, So-

mayaji et al. [51] delay anomalous system calls with increasing delay durations to prevent security violations such as a buffer overflow. Feinstein et al. [25] dynamically apply filtering rules when a DDoS attack is detected. Balepin et al. [14] use a cost model to compare predefined exploit costs with various predefined benefits to select the best response to a compromised system resource. Garfinkel et al. [29] sandbox applications by using user defined policies to control access to system resources by blocking certain system calls. Additionally, their tool emulates access to these resources to prevent applications from crashing. In contrast, our approach implements a fine-grained attack containment scheme that can reduce the disturbance to the user from the anomaly responses without using user defined detection or response rules.

6. CONCLUSION

In this paper, we have presented PREC, a novel classified system call monitoring and root exploit containment system for Android. PREC provides an on-device malware detection and containment solution to stop those malicious applications that hide themselves during market dynamic malware analysis using logic bombs or checks for an emulated environment. PREC achieves high detection accuracy and low false alarm impact by 1) classifying system calls by origin (e.g., third-party native library), 2) adapting SOM learning algorithm to detect root exploit activities, and 3) inserting exponentially increasing delays to malicious threads. We have implemented a prototype of PREC and evaluated it on 140 most popular benign applications and 10 malicious applications. Our experiments show that PREC successfully detected and stopped all the tested root exploit attacks. Compared to other anomaly detection algorithms, PREC reduces the false alarm rate by more than one order of magnitude. Our delay-based containment scheme only impose noticeable impact to 1 out of 140 popular benign applications. PREC is light-weight, which makes it practical for runtime malware containment on smartphone devices.

Acknowledgements

This work was supported in part by an NSA Science of Security Lablet grant at North Carolina State University, U.S. Army Research Office (ARO) under grant W911NF-10-1-0273, NSF CAREER Award CNS-1149445, NSF CAREER Award CNS-1253346, NSF grant CNS-1222680, IBM Faculty Awards, and Google Research Awards. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies. We would also like to thank Adwait Nadkarni, Ashwin Shashidharan, Vasant Tendulkar, Hiep Nguyen, and the anonymous reviewers for their valuable feedback during the writing of this paper.

7. REFERENCES

[1] Android Security Overview. Android Source. http://source.android.com/devices/tech/security/.

[2] Antutu Benchmark. https://play.google.com/store/apps/details?id=com.antutu.ABenchMark.

[3] DTrace. http://docs.oracle.com/javase/6/docs/technotes/guides/vm/dtrace.html.

[4] DWARF Debugging Standard. http://www.dwarfstd.org/.

[5] Exception Handling ABI for ARM Architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0038a/IHI0038A_ehabi.pdf.

[6] Ice Cream Sandwich. Android Developer. http://developer.android.com/about/versions/android-4.0-highlights.html.

[7] Linux man page - pTrace - process trace. http://linux.die.net/man/2/ptrace.

[8] Linux Trace Toolkit - next generation. https://lttng.org.

[9] Security-Enhanced Linux. Android Developer. http://source.android.com/devices/tech/security/se-linux.html.

[10] SystemTap. http://sourceware.org/systemtap/.

[11] UI/Application Exerciser Mokey. http://developer.android.com/tools/help/monkey.html.

[12] Vulnerabilities. X-Ray. http://www.xray.io/#vulnerabilities.

[13] Apple. Apple Updates iOS to 6.1. Apple. http://www.apple.com/pr/library/2013/01/28Apple-Updates-iOS-to-6-1.html.

[14] I. Balepin, S. Maltsev, J. Rowe, and K. Levitt. Using specification-based intrusion detection for automated response. In *Proc. of RAID*, 2003.

[15] U. Bayer, P. Milani, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Proc. of NDSS*, 2009.

[16] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995.

[17] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-Based Malware Detection System for Android. In *Proc. of CCS-SPSM*, 2011.

[18] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. A Quantitative Study of Accuracy in System Call-Based Malware Detection. In *Proc. of ISSTA*, 2012.

[19] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. MAST: Triage for Market-scale Mobile Malware Analysis. In *Proc. of WiSec*, 2013.

[20] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. In *Proc. of DSN*, 2008.

[21] J. Cheng, S. H. Wong, H. Yang, and S. Lu. SmartSiren: Virus Detection and Alert for Smartphones. In *Proc. of MobiSys*, 2007.

[22] G. Creech and J. Hu. A Semantic Approach to Host-based Intrusion Detection Systems Using Contiguous and Discontiguous System Call Patterns. *IEEE Transactions on Computers*, 2013.

[23] D. Dean, H. Nguyen, and X. Gu. UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *Proc. of ICAC*, 2012.

[24] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A Survey on Automated Dyanmic Malware-Analysis Techniques and Tools. *ACM Computing Surveys*, 44(2), Feb. 2012.

[25] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred. Statistical approaches to ddos attack detection and response. In *Proc. of the DARPA Information Survivability Conference and Exposition*, 2003.

[26] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proc. of IEEE Symposium on Security and Privacy*, 1996.

[27] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *Proc. of CCS*, 2004.

[28] D. Gao, M. K. Reiter, and D. Song. Behavioral Distance for Intrusion Detection. In *Proc. of RAID*, 2005.

[29] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proc. of NDSS*, 2003.

[30] L. Girardin and D. Brodbeck. A visual approach for monitoring logs. In *Proc. of LISA*, 1998.

[31] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proc. of MobiSys*, 2012.

[32] W. Hu, Y. Liao, and V. R. Vemuri. Robust anomaly detection using support vector machines. In *Proc. of ICML*, 2003.

[33] T. Isohara, K. Takemori, and A. Kubota. Kernel-based Behavior Analysis for Android Malware Detection. In *Proc. of CIS*, 2011.

[34] H. Jiang and J. Ruan. The application of genetic neural network in network intrusion detection. *Journal of Computers*, 2009.

[35] T. Kohonen, J. Tan, and T. Huang. *Self-Organizing Maps*. Springer, 3rd edition, 2001.

[36] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *Proc. of USENIX Security*, 2009.

[37] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proc. of ESORICS*, 2003.

[38] P. Lichodzijewski, A. Nur Zincir-Heywood, and M. Heywood. Host-based intrusion detection using self-organizing maps. In *Proc. of IJCNN*, 2002.

[39] H. Lockheimer. Android and Security. Google Mobile Blog. http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[40] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE TODS*, 2008.

[41] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *Proc. of AST*, 2013.

[42] C. Michael and A. Ghosh. Using Finite Automata to Mine Execution Data for Intrusion Detection: A Preliminary Report. In *Proc. of RAID*, 2000.

[43] A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proc. of IEEE Symposium on Security and Privacy*, 2007.

[44] J. Oberheide. Dissecting Android's Bouncer. The Duo Bulletin. https://blog.duosecurity.com/2012/06/dissecting-androids-bouncer/.

[45] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-Version Antivirus in the Network Cloud. In *Proc. of USENIX Security*, 2008.

[46] C. Peng, C. yu Li, G. hua Tu, S. Lu, and L. Zhang. Mobile Data Charging: New Attacks and Countermeasures. In *Proc. of CCS*, 2012.

[47] H. Pilz. Building a Test Environment for Android Anti-malware Tests. In *Proc. of VB*, 2012.

[48] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid Android: Versatile Protection For Smartphones. In *Proc. of ACSAC*, 2010.

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

[50] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: a perspective combining risks and benefits. In *Proc. of SACMAT*, 2012.

[51] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proc. of the USENIX Security*, 2000.

[52] A. M. R. Tahiliani and M. Naik. Dynodroid: An input generation system for android apps. Technical report, Georgia Institute of Technology, 2012.

[53] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusions Using System Calls: Alternative Data Models. In *Proc. of IEEE Symposium on Security and Privacy*, 1999.

[54] B. Womack. Google Says 700,000 Applications Available for Android. Bloomberg Businessweek, Oct. 2012. http://www.businessweek.com/news/2012-10-29/google-says-700-000-applications-available-for-android-devices.

[55] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu. pBMDS: A Behavior-based Malware Detection System for Cellphone Devices. In *Proc. of WiSec*, 2010.

[56] W. Yang, M. Prasad, and T. Xie. A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications. In *Proc. of FASE*, 2013.

[57] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proc. of IEEE Symposium on Security and Privacy*, 2009.

[58] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proc. of CCS-SPSM*, 2012.

[59] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proc. of IEEE Symposium on Security and Privacy*, 2012.

[60] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proc. of NDSS*, 2012.