ABSTRACT

HO, TSUNG-HSUAN. PREC: Practical Root Exploit Containment for Android Devices. (Under the direction of Dr. Xiaohui (Helen) Gu and Dr. William Enck.)

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 the 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 which protects users from this type of 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. This allows PREC to detect and stop root exploits with high accuracy without interfering with benign applications. We have implemented PREC and evaluated our methodology using 140 of the most popular benign applications along with 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 an 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.
DEDICATION

To my parents and my wonderful girlfriend Liheng.
BIOGRAPHY

The author has completed his B.S. degree in Computer Science and Engineering in 2006 from National Sun Yat-sen University, Taiwan and M.S. degree in Information Systems and Applications in 2008 from National Tsing Hua University respectively. He then joined CyberLink for smartphone and multimedia development. In fall 2011, he returned to school joining NC State University as a Ph.D. student in Computer Science Department. He transferred to M.S. track in 2013. After completion of his master degree, he will be joining Citrix.
I would like to thank Dr. Xiaohui Gu and Dr. William Enck for being great advisors and mentors. This thesis would not have been possible without their guidance and support. I would like also thank Dr. Min Chi for taking the time to be on my thesis committee and for being supportive to my research. I would like especially thank Daniel Dean for his tremendous help to this thesis. I also thank my peers Al Gorski, Chin-Jung Hsu, Adwait Nadkarni, Hiep Nguyen, Ashwin Shashidharan, and Vasant Tendulkar for their valuable inputs on this work.
# TABLE OF CONTENTS

**LIST OF TABLES** ................................................................. vi

**LIST OF FIGURES** ................................................................ vii

**Chapter 1 Introduction** ......................................................... 1
  1.1 Motivation ................................................................. 1
  1.2 Summary of the State of the Art .................................. 2
  1.3 Proposed Solution ..................................................... 3
  1.4 Thesis Statement ....................................................... 4
  1.5 Contributions .......................................................... 5
  1.6 Assumptions ............................................................. 5
  1.7 Summary of Experiment Results .................................. 6
  1.8 Thesis Organization .................................................. 7

**Chapter 2 Overview** .............................................................. 8

**Chapter 3 System Design and Prototype Implementation** ............. 12
  3.1 System Call Origin Identification ................................ 12
  3.2 On-Device Root Exploit Detection .............................. 15
    3.2.1 Normal app behavior learning ............................. 15
    3.2.2 Use of system call arguments ............................. 18
    3.2.3 Runtime root exploit detection ............................ 19
  3.3 Automatic Root Exploit Containment ............................ 19

**Chapter 4 Experimental Evaluation** ........................................ 21
  4.1 Evaluation Methodology ............................................. 21
  4.2 Results and Analysis ................................................ 27

**Chapter 5 Related Work** .................................................... 42

**Chapter 6 Discussion** ........................................................... 46

**Chapter 7 Conclusion** ......................................................... 48

**References** ......................................................................... 49
LIST OF TABLES

Table 4.1 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. ............................................... 22

Table 4.2 Anomaly detection model training time comparison. The experiment was conducted on a CentOS 6.2 Linux server with a 2.5GHz Intel XEON X3440 quad-core processor (HyperThreading disabled) and 8GB of physical memory. The average per-app system call sequence number is 244K under all system call monitoring and 106K under native thread system call monitoring. ................................................................. 34

Table 4.3 Malware detection and containment results. ....................................................... 37

Table 4.4 Delay-based containment impact to benign apps with false alarms. Each application run lasts three minutes. ................................................................. 39

Table 4.5 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. “$S = \text{max}$” means the number of states equal to the number of distinctive system calls in the trace. The experiment was conducted on a Ubuntu 12.04LTS Linux desktop with a 3.4GHz Intel Core i7-2600 quad-core processor (HyperThreading disabled) and 8GB of physical memory. The average per-app system call sequence number is 244K under all system call monitoring and 106K under native thread system call monitoring. 40
Figure 2.1 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. 9

Figure 3.1 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. 13

Figure 3.2 SOM update example using the input vector [1,2,4]. 16

Figure 4.1 Percentage of system calls originated from native code for 10 malicious apps. 27

Figure 4.2 Percentage of system calls originated from native code for 80 apps with native code. 28

Figure 4.3 Per-app malware detection accuracy comparison results for 80 apps with native code. Detection was performed without consideration of arguments. 30

Figure 4.4 Percentage of system calls originated from native code for 80 apps with native code. Detection was performed with consideration of arguments. 30

Figure 4.5 Per-app malware detection accuracy comparison results for 60 apps that do not have any native code. Detection was performed without consideration of arguments. 31

Figure 4.6 Per-app malware detection accuracy comparison results for 60 apps that do not have any native code. Detection was performed with consideration of arguments. 31

Figure 4.7 Per-sequence false positive rate comparison for 80 apps that include native code. Detection was performed without consideration of arguments. 32

Figure 4.8 Per-sequence false positive rate comparison for 80 apps that include native code. Detection was performed with consideration of arguments. 32

Figure 4.9 Per-sequence false positive rate comparison for 60 apps that do not include any native code. Detection was performed without consideration of arguments. 33
Figure 4.10 Per-sequence false positive rate comparison for 60 apps that do not include any native code. Detection was performed with consideration of arguments. ........................................... 33

Figure 4.11 Per-app malware detection accuracy comparison (with or without cross-validation) results for 80 apps with native code. Detection was performed without consideration of arguments. .................. 35

Figure 4.12 Per-app malware detection accuracy comparison (with or without cross-validation) results for 80 apps with native code. Detection was performed with consideration of arguments. .................. 35

Figure 4.13 Per-sequence false positive rate comparison (with or without cross-validation) for 80 apps that include native code. Detection was performed without consideration of arguments. (CV: cross validation) 36

Figure 4.14 Per-sequence false positive rate comparison (with or without cross-validation) for 80 apps that include native code. Detection was performed with consideration of arguments. (CV: cross validation) 36

Figure 4.15 PREC runtime performance overhead under different benchmark apps on Galaxy Nexus running Android 4.2. .......................... 41
Chapter 1

Introduction

1.1 Motivation

The rise in popularity of smartphones can be greatly attributed to the vast and diverse collection of available third-party applications. Popular application markets (e.g., Apple’s App Store, and the Google Play Store) currently contain over 700,000 applications each [13, 55]. Markets 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 [40]), dynamic analysis environments can be easily detected by malware through methods such as checking certain system properties or IP addresses [20, 23, 45]. Once an analysis environment is detected, the malware can simply mask its malicious activities by using logic bombs or by disabling the attack. Current mitigation solutions include hiding or removing emulated features commonly detected by malware and using static analysis to identify all possible code paths [23]. The latter is expensive and can be easily perturbed using code obfuscation.

We propose a novel approach to mitigating malware that is attempting to hide from
market malware analysis. Our key observation is that dynamic malware analysis performed by application markets provides an opportunity to learn a normal behavior profile for an application. We then use runtime system call anomaly detection and malware containment to protect the user from malicious activities (e.g., exploiting a root privilege escalation vulnerability). In effect, this forces application authors to commit to the behavior during market malware analysis. Therefore, zero-day malware that uses logic bombs or other techniques to change execution based on an emulated environment will be detected on the smartphone device at runtime. Our approach is designed specifically to target malware that tries to evade powerful dynamic malware analysis which can detect any suspicious system state changes using more heavyweight techniques such as virtual machine introspection [57].

1.2 Summary of the State of the Art

To develop a normal execution profile for each application, we use system call sequence monitoring. System call based intrusion detection system (IDS) was first introduced by Forrest et al. [25] in 1996. There are two general system call sequence monitoring approaches used by IDS to detect malicious code. The first approach, called signature-based detection, learns the specific sequence of system calls or patterns corresponding to the malicious code. Recent work such as CloudAV [46] and DroidRanger [62] falls into this category. While signature-based detection is very good at identifying known malicious sequences, it cannot detect zero-day malware. It is also difficult for such signature based approaches to detect mimicry attacks. The second approach, called anomaly-based detection, tries to learn the normal system call sequences generated by the application as opposed to specific malicious sequences. The idea behind this approach is that if the IDS
knows the normal behavior, malicious code should be detected as abnormal behavior. Forrest’s work [25] is an example of this approach. Their approach characterizes system call sequences into fixed-length \textit{ngram} patterns that are stored in a database. Any behavior that deviates from these patterns is determined malicious. While this approach can usually detect malicious behaviors, it suffers from false positives and is vulnerable to mimicry attacks. These two aspects are at odds. Reducing false positives often requires making the detection algorithm less sensitive to change; however, this simultaneously makes the algorithm more susceptible to mimicry attacks. Hence, several algorithms have been introduced to improve the precision and the accuracy of anomaly-based intrusion detection systems. For example, Warrender et al. [54] uses hidden Markov models (HMMs) and previous work has adapted algorithms from artificial neural network (ANN) [16, 35, 22]. Although these efforts can achieve reasonably low false positive rates, we observe that existing approaches raise false alarms for most of the tested applications. We call this kind of false alarms \textit{per-application false positives}.

### 1.3 Proposed Solution

To this end, we present the Practical Root Exploit Containment (PREC) framework for the Android platform. PREC adapts anomaly-based approach to detect root privilege escalation attacks due to the ability of the anomaly-based approach to find zero-day malware, along with the benefit of not requiring any malicious sequences. Like many similar anomaly detection systems, the primary challenge of making PREC practical is achieving a low false alarm rate [18] to minimize the interruption of using benign applications. PREC addresses this challenge by using two novel techniques.

First, PREC uses \textit{classified system call monitoring}, which separates system calls based
on their origins (e.g., the library that initiates the system call). Specifically, PREC focuses on system calls originating in third-party native libraries to defend against malware that attempts to gain root privilege of the system [32]. The intuition behind this is that we observed that existing root privilege escalation attacks are released and performed through native libraries. Therefore, we focus on the possibly malicious component only to enhance to detection accuracy.

To stop detected potential attacks, we propose two novel containment approaches. The first one is called thread-based termination containment. Unlike traditional ways where the system kills the malicious process when malicious behavior is detected, we only terminate the malicious thread when PREC raises an alarm. The rationale behind this mechanism is that the malicious code is usually planted in a benign app. Simply terminating the malicious thread will not affect the execution of the normal part. Another mechanism we use is called delay-based containment. When PREC raises alarms, it delays the system call for a period of time. The delay period increases exponentially and decreases linearly. The intuition behind this mechanism is that we observe malicious attacks usually need to swamp a large number of system calls in a short time, and delaying the system call execution can effectively stop the attack. Normal applications, on the other hand, only have some isolated false alarms, and therefore will not be affected by this approach.

1.4 Thesis Statement

“Monitoring system calls via origin classification and using delay-based containment improve the detection and the prevention of root privilege escalation attacks in Android.”
1.5 Contributions

This thesis 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 behavior during 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 for higher anomaly detection accuracy and more practical malware containment.

- We provide a scalable and robust behavior learning and anomaly detection scheme using the self-organizing map (SOM) learning algorithm [36] that can achieve both high accuracy and low overhead.

- We propose a novel delay-based fine-grained approach to contain possible root privilege attacks while minimizing the false alarm impact to benign applications.

1.6 Assumptions

Our work makes a set of assumptions that are summarized as follows.

- We assume that root escalation attacks can only be performed through native libraries. Although it is possible for a root privilege escalation attack to originate from Java code, exploits usually require access to low-level APIs which are difficult to execute purely in Java. As a result, there is no known existing root exploits originating from Java code. While this type of attack may seem very limited in
scope, we argue that the protection provided by PREC is extremely valuable in practice because once the malware has root, it 1) can perform nearly any action, 2) is extremely difficult to remove, and 3) cannot be easily detected with existing malware analysis techniques based on permissions [51, 47, 19].

- We also assume that the in-market detection system can detect any malicious behaviors that can possibly be used to perform root privilege escalation attacks. This is a reasonable assumption because offline detection system can use powerful detection methods to detect possible attacks from the applications.

- We also assume that the in-market detection system can provide good coverage of the application behavior.

1.7 Summary of Experiment Results

We have implemented PREC and evaluated our methodology on 140 of the 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 [61] 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 practicality 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 in the whole market have native code [32], we expect the false alarm rate for PREC will be very low in practice. Cross validation does not improve our per-app false positive rates but posts a huge amount of overhead to the training phase. Therefore we did not use cross validation in our prototype. 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 only 1 out of 140 tested popular benign applications. PREC is light-weight and only imposes less than 3% runtime execution overhead on the smartphone device.

1.8 Thesis Organization

This thesis is organized as follows. Chapter 2 gives an overview about the system. Chapter 3 presents the detailed design and prototype implementation of the system. In Chapter 4, we explain the experiment setup and discuss the experimental results. Chapter 5 provides a survey about the related work. Chapter 6 discuss the assumptions and limitations. Chapter 7 concludes this thesis.
Chapter 2

Overview

The Practical Root Exploit Containment (PREC) framework extends the existing mobile application market model by providing practical online root exploit detection and containment for smartphone users. Figure 2.1 depicts the overall PREC architecture. PREC operates in two phases: 1) PREC Learning: offline training when a developer submits an app into the market; and 2) PREC Enforcement: 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 will reject it. 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 [20, 23, 45]. This is where PREC provides its 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 example, PREC
Figure 2.1: 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.

labels each system call as originating either from third-party native code or from Java code. This labeled trace is then used by PREC to create a normal behavior model for the app. The normal behavior model is sent to the PREC service which could be hosted within a computing cloud.

During dynamic malware analysis, PREC records and labels a system call trace based on our classified monitoring criteria. For example, PREC labels each system call as originating either from third-party native code or from Java code. This labeled trace is then used by PREC to create a normal behavior model for the app. The normal behavior model is sent to the PREC service which could be hosted within a computing cloud (e.g., Amazon EC2). 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) [36]. The SOM was designed to project a high dimensional input space onto a low dimensional output space. We chose the SOM because it is resilient to noise and is significantly less computation-intensive than other approaches such as clustering.

Ideally, the normal behavior model should be comprehensive in order to avoid false alarms. As a result, PREC allows developers to submit an optional input trace to collect a precise normal execution profile. Dynamic analysis of smartphone apps is currently a significant challenge for mobile app researchers [29]. Recent research [12] has shown that submitting an input trace allows developers to reduce the possibility of false positives in the enforcement phase. More specifically, PREC generates a normal profile by following an input trace when an input trace is available. If an input trace is not provided, more traditional test case generation techniques or random input fuzz-testing [31] can be used. The current prototype of PREC uses a simple random input fuzz testing tool [10]. Our experiments show that this simple approach can produce high quality behavior models for most of the real Android apps. However, PREC is a general behavior learning framework, which can be integrated with any input testing tools.

The enforcement phase uses the normal behavior model from the PREC service to perform on-device anomaly detection. We specifically focus on malware that exploits root privilege escalation vulnerabilities. This type of malware represents the highest risk for smartphones, because it allows for the greatest amount of malicious functionality, can hide its existence, and is difficult to remove [32]. All existing malware exploiting root privilege escalation vulnerabilities have done so using third-party native code, whether it be a library invoked via the Java native interface (JNI), or a stand-alone executable. Therefore, PREC performs anomaly detection only on system calls that originate in
third-party native code. Monitoring only third-party native code significantly reduces false alarms of runtime root privilege escalation attack detection.

PREC further reduces the impact of false alarms by intelligent anomaly containment. In order to classify system calls originating from third-party native libraries, PREC executes calls to native libraries in a separate native thread. Therefore, when an anomaly occurs, PREC can isolate the infringing thread. At this point, PREC can do one of two things. The first option is to simply kill the native thread. The intuition behind this approach is that malware authors usually graft an attack onto a benign application. Killing just the anomalous native thread will not affect the execution of the benign components. PREC’s second option is to slow the thread down by inserting an exponentially increasing artificial delay into system calls that increases whenever an anomaly occurs. Intuitively, anomalies should be infrequent for benign applications. However, exploiting vulnerabilities will almost always result in a burst of anomalies. The rationale behind our approach is that address space layout randomization (ASLR) in Android [5] enforces 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) are resource exhaustion attacks (e.g., continuously forking). By slowing down the malicious activity with a sufficient delay, the exploit becomes invalid or cause an Application Not Responding (ANR) event to kill the malicious app.
Chapter 3

System Design and Prototype 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.

3.1 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 another option to understand program components on the call path.
Figure 3.1: 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.

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 [3] or EXIDX [4]) that is needed to unwind the stack. Therefore, 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 3.1.

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 [6]) that can introduce over 20% overhead due to frequent context switches, our kernel tracer only incurs less than 2%. PREC could also use other kernel space system call tracing tools such as SystemTap [9], DTrace [2], or Linux Trace Toolkit - next generation (LTTng) [7] 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 thesis, 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 contain a small portion of the application rather than kill the whole application. This allows us to minimize the containment scope (i.e., reducing the disturbance to the user) when the anomaly detection raises an alarm. We will describe our containment scheme in Section 3.3. Third, PREC can easily incorporate other execution sandboxing mechanisms (e.g., software fault isolation [59]) to provide additional security isolations between Java code and malicious native code.

3.2 On-Device Root Exploit Detection

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

3.2.1 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 [36]. 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. SOM preserves the topological properties of the original input space (i.e., two similar samples will be projected to close positions in the map). In our system, each
The SOM map consists of \( n \times m \) nodes called neurons arranged in a grid, illustrated by Figure 3.2. 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 \( 1 \leq i \leq S \), and \( S \) is equal 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 (\texttt{sys_chdir}) is completely different than system call id 13 (\texttt{sys_time}). To address these issues, we have made two modifications to the traditional SOM learning algorithm. First, we use the graph 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 3.2. First, we form an input vector of length \( k \) by reading \( k \) system calls from the training data. Second, we examine the graph 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 3.2, 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 3.2, 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.

### 3.2.2 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).
3.2.3 Runtime root exploit detection.

When a user purchases an app from the market, the SOM model 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 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.

3.3 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 killing or slowing down the malicious native threads only instead of the whole application.

PREC enables dynamic native thread termination by inserting a signal handler inside the native thread before the native function is called. When an alarm is raised, the
anomaly detection model sends a predefined signal to the malicious native thread to terminate the thread. 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 native thread by inserting a delay during the native thread 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 application will not suffer from crashing due to the thread killing.

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. 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 anomaly. Our prototype starts at 1 ms and doubles per anomaly. There are many potential policies for delay decrease. Our prototype currently exponentially decreases (halves) for each non-anomalous system call. We also considered a linear decrease, but found exponential decrease to handle more false alarms. Both the delay increase and decrease policies are knobs for tuning false alarm sensitivity.
Chapter 4

Experimental Evaluation

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

4.1 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) to use as benign applications. 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 native code and majority of them are games and multimedia applications. We also test 60 popular apps without any native code. We use more benign apps with native code than without native code in order to estimate the worst-case false
Table 4.1: 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.

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

alarm rate of PREC since PREC will not raise any false alarm for benign apps without any native code. In contrast, other alternative schemes without our classified monitoring techniques will still raise false alarms on those benign apps without 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 4.1 shows the 10 malicious applications used in our experiments that covers four real root exploits. We first used four real malware reported by the Malware Genome project [61]. To evaluate PREC under more challenging cases, we repackage existing root privilege escalation attacks into a popular application (AngryBirds) that contains a lot of native code executions. We
believe our malware coverage is extensive, which is explained as follows.

We first studied all the six root exploit malware families (DroidDream, DroidKongFu1, DroidKongFu2, Ginger Master, BaseBridge, DroidKungFuSapp) reported by the Malware Genome Project [61]. 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, we found this server is already down at the time of our testing, which disallows us to trigger the root exploit.

The RiskRanker project [32] and the X-ray project [11] 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 (ADB) to determine whether developer can login as root. However, attackers cannot embed this exploit into applications because Android applications cannot access ADB. Similarly, zergRush requires several information in ADB and Mempodroid executes run-as inside the Android Debug Bridge shell. Therefore, it is feasible for attackers to utilize those exploits in applications. The rest two root exploits (Asroot or named Wunderbar in X-ray, 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, we believe that root privilege escalation attack is an increasing concern in Android. For example, Google introduced SELinux in Android 4.3 to mitigate the damage of root escalation attacks [8]. 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 [10] 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 [53] 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 6. 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 [60, 58, 50, 42] 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 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 employ 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 4.1, 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 [54] over the system calls from the native code only, which learn normal system call sequence transition prob-
abilities and raises an alarm if the observed tradition probability is below a threshold; 3) \textit{HMM (full)} that uses the hidden Markov model over all system calls; 4) \textit{FSA (native)} [43] that uses the finite state automaton over the system calls from the 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) \textit{FSA (full)} that uses a finite state automaton over all system calls. Note that we only compare PREC with common \textit{unsupervised} learning methods since supervised learning methods (e.g., support vector machine [33]) 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 \textit{true positive rate} $A_T$ and \textit{false positive rate} $A_F$ metrics, as shown in Equation 4.1. $N_{tp}$, $N_{fn}$, $N_{fp}$, and $N_{tn}$ denote the true positive number, false negative number, false positive number, and true negative number, respectively.

$$A_T = \frac{N_{tp}}{N_{tp} + N_{fn}}, \quad A_F = \frac{N_{fp}}{N_{fp} + N_{tn}} \tag{4.1}$$

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.
4.2 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 4.1 shows the percentage of the system calls originated from the native code for the 80 benign apps that include native code. Although all those 80 apps contain native code, we observe that over 50% of the apps execute less than 10% native code. Thus, PREC can still filter out a large number of system calls for those benign applications with native code during model creation and malware detection. We also observe that PREC never misclassifies a system call from Java as a system call from native code. Thus, PREC will not raise any false alarm for those benign applications that do not include any native code. Figure 4.2 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 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.

![Figure 4.1: Percentage of system calls originated from native code for 10 malicious apps.](image)
Figure 4.2: Percentage of system calls originated from native code for 80 apps with native code.

**Runtime on-device detection accuracy:** We now evaluate the runtime on-device detection accuracy of the PREC system. Figure 4.4 and Figure 4.3 shows the per-app true positive and false positive rate using different anomaly detection algorithms for the 80 benign apps that include native code. Figure 4.3 shows the results without considering system call arguments while Figure 4.4 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 threshold, 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 SOM is more robust to noise in system calls than HMM and FSA because it projects 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 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 native code. Figure 4.5 and Figure 4.6 shows the anomaly detection accuracy results of different algorithms without using the classified system call monitoring scheme for 60 benign apps without 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 4.7 and Figure 4.8 shows the per-sequence false positive rates at 85-99 percentile achieved by different schemes for the 80 benign apps that include 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 4.9 and Figure 4.10 shows the per-sequence false positive rate comparison for the benign apps without any native code. We also observe that SOM can significantly reduce the false
Figure 4.3: Per-app malware detection accuracy comparison results for 80 apps with native code. Detection was performed without consideration of arguments.

Figure 4.4: Percentage of system calls originated from native code for 80 apps with native code. Detection was performed with consideration of arguments.
Figure 4.5: Per-app malware detection accuracy comparison results for 60 apps that do not have any native code. Detection was performed without consideration of arguments.

Figure 4.6: Per-app malware detection accuracy comparison results for 60 apps that do not have any native code. Detection was performed with consideration of arguments.
positive rate by orders of magnitude.

Figure 4.7: Per-sequence false positive rate comparison for 80 apps that include native code. Detection was performed without consideration of arguments.

Figure 4.8: Per-sequence false positive rate comparison for 80 apps that include native code. Detection was performed with consideration of arguments.
We also evaluate the possible impact that cross validation can bring to our system. Cross validation is a method to obtain the possibly best random number for the ma-
chine learning algorithms that require a random number to initiate the system. In this experiment, we use a three-fold cross validation with four different random numbers. A three-fold cross validation divides the training data into three pieces to perform three rounds of tests. In each round, we pick two pieces for training and one piece for the validation test. We repeat the process until all three pieces have been selected as the tested piece. After three rounds of tests, we can have a best result out of three results generated from the three rounds of tests. We then use the random number to evaluate the performance of SOM.

As we can see in Figure 4.11 and Figure 4.12, cross validation does not affect our per-app result very much. This is because once the SOM has been well trained with the sufficient data, the map should be able to represent all behaviors of the application, and therefore, it does not affect the result too much. Figure 4.13 and Figure 4.14 show the deeper results of SOM: per-sequence results. These two figures show higher difference between with and without cross validation for per-sequence results. This is because we have achieved very small false positive rates, any small fluctuation can vary the result big change to the average shown in Figure 4.13 and Figure 4.14. Table 4.2 shows the overhead of using cross validation. Cross validation increase training time 26-28 times. Given that cross validation only improves marginally but brings huge overhead, most of the experiments performed in this thesis do not enable the function of cross validation.

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 4.3 summarizes our malware containment results. As mentioned in Section 3.3, PREC provides two different containment mechanisms: 1) termination-based containment that stops the
Figure 4.11: Per-app malware detection accuracy comparison (with or without cross-validation) results for 80 apps with native code. Detection was performed without consideration of arguments.

Figure 4.12: Per-app malware detection accuracy comparison (with or without cross-validation) results for 80 apps with native code. Detection was performed with consideration of arguments.
Figure 4.13: Per-sequence false positive rate comparison (with or without cross-validation) for 80 apps that include native code. Detection was performed without consideration of arguments. (CV: cross validation)

Figure 4.14: Per-sequence false positive rate comparison (with or without cross-validation) for 80 apps that include native code. Detection was performed with consideration of arguments. (CV: cross validation)
Table 4.2: Anomaly detection model training time comparison. The experiment was conducted on a CentOS 6.2 Linux server with a 2.5GHz Intel XEON X3440 quad-core processor (Hyper-Threading disabled) and 8GB of physical memory. The average per-app system call sequence number is 244K under all system call monitoring and 106K under native thread system call monitoring.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>per-app training time without cross validation</th>
<th>per-app training time with cross validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREC</td>
<td>158.2 ± 259.9s</td>
<td>4232.6 ± 7918.7s</td>
</tr>
<tr>
<td>SOM (full)</td>
<td>343.2 ± 291.1s</td>
<td>9827.3 ± 10648.9s</td>
</tr>
</tbody>
</table>

Table 4.3: Malware detection and containment results.

<table>
<thead>
<tr>
<th>Malware Samples</th>
<th>Alarm lead time</th>
<th>Termination-based containment</th>
<th>Delay-based containment</th>
</tr>
</thead>
<tbody>
<tr>
<td>DroidDream</td>
<td>20.7 sec</td>
<td>success</td>
<td>success</td>
</tr>
<tr>
<td>DroidKungFu1</td>
<td>16.1 sec</td>
<td>success</td>
<td>success</td>
</tr>
<tr>
<td>DroidKungFu2</td>
<td>96.5 sec</td>
<td>success</td>
<td>success</td>
</tr>
<tr>
<td>GingerMaster</td>
<td>318.3 sec</td>
<td>success</td>
<td>success</td>
</tr>
<tr>
<td>RATC-1</td>
<td>26.6 sec</td>
<td>success</td>
<td>success</td>
</tr>
<tr>
<td>RATC-2</td>
<td>17.1 sec</td>
<td>success</td>
<td>success</td>
</tr>
<tr>
<td>ZimperLich-1</td>
<td>14.1 sec</td>
<td>success</td>
<td>success</td>
</tr>
<tr>
<td>ZimperLich-2</td>
<td>20.9 sec</td>
<td>success</td>
<td>success</td>
</tr>
<tr>
<td>GingerBreak-1</td>
<td>35 sec</td>
<td>success</td>
<td>success</td>
</tr>
<tr>
<td>GingerBreak-2</td>
<td>34.6 sec</td>
<td>success</td>
<td>success</td>
</tr>
</tbody>
</table>

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 root exploit is detected and 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 \texttt{[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 \texttt{chmod} executable to change permission to \texttt{0755} for later execution. Because different devices place \texttt{chmod} in different directories, the root exploit needs to try several locations to find the right directory. For DroidDream, the detected anomalous sequence is \texttt{[execve, execve, execve, execve, close, execve, read, close, mprotect]}]. In this case, DroidDream first execute the \texttt{explode} exploit to try to compromise the system. The first detected alarm for DroidKungFu1 is \texttt{[writev, gettid, writev, writev, writev, pipe, getrlimit, close, ustat]}]. This sequence contains several system calls that used by the RATC attack. \texttt{Pipe} is used for the communication between attack parent process and child processes. The system call \texttt{getrlimit} is used to understand the limitation on the number of processes for the current user. DroidKungFu2 is detected with the sequence \texttt{[execve, close, getpid, sigaction, sigaction, sigaction, sigaction, sigaction, sigaction]}]. The malware uses the \texttt{execve} system call to execute the built-in root exploit \texttt{secbino}. Both ZimperLich-1 and ZimperLich-2 are caught by PREC with the sequence \texttt{[sigaction, sigaction, sigaction, sigaction, getuid, geteuid, getgid, getegid, stat]}]. The system calls from the 5'th position to the 9'th position are exactly the first five system calls used in the ZimperLich source code. Those sequence analysis results show that PREC accurately catches the malicious behaviors of those malware samples.

**False alarm impact results:** We now evaluate the false alarm impact of our system
Table 4.4: Delay-based containment impact to benign apps with false alarms. Each application run lasts three minutes.

<table>
<thead>
<tr>
<th>Application</th>
<th>Fine-grained containment</th>
<th>Whole app containment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forthblue.pool</td>
<td>Hang</td>
<td>Hang</td>
</tr>
<tr>
<td>TalkingTom2Free</td>
<td>0.1 sec</td>
<td>Hang</td>
</tr>
<tr>
<td>CamScanner</td>
<td>0.25 sec</td>
<td>Hang</td>
</tr>
</tbody>
</table>

using different containment schemes. As shown in Figure 4.3, Figure 4.4, Figure 4.5, and 4.6, 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 4.5 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 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
Table 4.5: 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. “\( S = \text{max} \)” means the number of states equal to the number of distinctive system calls in the trace. The experiment was conducted on a Ubuntu 12.04LTS Linux desktop with a 3.4GHz Intel Core i7-2600 quad-core processor (HyperThreading disabled) and 8GB of physical memory. The average per-app system call sequence number is 244K under all system call monitoring and 106K under native thread system call monitoring.

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

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 [1]. Figure 4.15 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.
Figure 4.15: PREC runtime performance overhead under different benchmark apps on Galaxy Nexus running Android 4.2.
Chapter 5

Related Work

Forrest et al. [25] first proposed the system call malware detection schemes by building a database of normal system call sequences. Warrender et al. [54] extended this idea by using hidden Markov models (HMMs) to model sequences of normal system calls. Other researchers [16, 35, 22] adapt artificial neural network to perform intrusion detection. Kruegel et al. [38] proposed to use system call arguments to improve the performance of host-based intrusion detection. Maggi et al. [41] proposed to cluster similar system calls or similar system call arguments to further improve the accuracy. Previous work [30, 39] also used SOM for network intrusion detection by clustering system call arguments such as user name, connection type, and connection time. Gao et al. [27, 26] 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. SVM [33] and classic signature-based approaches are supervised learning methods, which require malware data during the learning phase and cannot detect unknown malware. In contrast, SOM is an unsupervised learning method, which does not require malware training data and can detect unknown malware.

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 [49] 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 incurs 15-30% overhead to smartphone devices. In contrast, PREC only imposes less than 5% 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. [34] monitor a pre-defined subset of system calls such as open on smartphones. pBMDS [56] 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. [44] 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. [37] generate hard to obfuscate models that track the dependencies of system calls in known malware, which they then can use to detect malware. CloudAV [46] 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 [62] 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 [51, 47, 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, Somayaji et al. [52] delay anomalous system calls by larger and larger amounts to prevent security violations such as a buffer overflow. Feinstein et al. [24] 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. [28] 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.
Chapter 6

Discussion

Classified system call monitoring reduces the monitor scope to system calls only from third party native libraries. However, it is indeed possible that attackers exploits the system through Java-level APIs. Although it is possible for an root privilege escalation attack to originate from Java code, we note that this approach is more difficult for attachers because of following reasons. First, exploits usually require access to low-level APIs which are difficult to execute purely in Java. As a result, there are no known existing root exploit originating from Java. Second, static analysis of Java-based code in Android applications is significant easier. A Java bytecode representation can be obtained (after retargeting from DEX), which retains significantly more semantics than ARM assembly. Therefore, it is significant easier for static analysis tools to detect a malware that mounts a root privilege escalation attack from Java code, particularly given the amount of Android malware program analysis currently occurring in industry and academic analysis [60, 58, 50, 42].

In addition, our current trace collection prototype implementation has two limitations. First, simple fuzz-testing tools such as the Monkey tool is not guaranteed to produce
sufficient behavioral coverage although 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 [50]. Developers can also provide application
input traces, which 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 [48].

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.

PREC proposes a delay-based approach to contain potential attacks. This approach
does not work for single- or few-sequence attacks. Fortunately, due to the dynamic nature,
complexity of the computer systems, and built-in protection schemes (e.g., Address Space
Layout Randomization, ASRL), all exploits we have studied require a fair number of
system calls to obtain sufficient information to perform attack or to exhaust a specific
system resource. However, this is a potential limitation of our system.
Chapter 7

Conclusion

In this thesis, 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.
REFERENCES


