Morpheus: Automatically Generating Heuristics to Detect Android Emulators

Yiming Jing[†], Ziming Zhao[†], Gail-Joon Ahn[†], and Hongxin Hu[‡]
[†]Arizona State University [‡]Clemson University
{ymjing,zmzhao,gahn}@asu.edu, hongxih@clemson.edu

ABSTRACT

Emulator-based dynamic analysis has been widely deployed in Android application stores. While it has been proven effective in vetting applications on a large scale, it can be detected and evaded by recent Android malware strains that carry detection heuristics. Using such heuristics, an application can check the presence or contents of certain artifacts and infer the presence of emulators. However, there exists little work that systematically discovers those heuristics that would be eventually helpful to prevent malicious applications from bypassing emulator-based analysis. To cope with this challenge, we propose a framework called Morpheus that automatically generates such heuristics. Morpheus leverages our insight that an effective detection heuristic must exploit discrepancies *observable* by an application. To this end, Morpheus analyzes the application sandbox and retrieves observable artifacts from both Android emulators and real devices. Afterwards, Morpheus further analyzes the retrieved artifacts to extract and rank detection heuristics. The evaluation of our proof-of-concept implementation of Morpheus reveals more than 10,000 novel detection heuristics that can be utilized to detect existing emulator-based malware analysis tools. We also discuss the discrepancies in Android emulators and potential countermeasures.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Emulators*

Keywords

Android, emulator, malware

1. INTRODUCTION

Recent years have witnessed an explosive growth of mobile applications. According to the Gartner report [18], there were 64 billion application downloads worldwide in 2012 and it was increased to 102 billion in 2013. The growth should partially give credit to application stores such as Apple App-Store and Google Play. With these services, users enjoy a centralized and trusted source for browsing and purchasing applications. Meanwhile, developers also get chances to reach a wider audience and more profits. Unfortunately, such advantages also make application stores an appealing place for distributing malicious mobile applications. To infect more unsuspecting users, adversaries would attempt to publish their malware in application stores without being detected.

To effectively mitigate such attempts, the security community and the application stores have deployed emulator-based dynamic analysis. Compared with static analysis that can be thwarted by obfuscation and encryption [15], emulator-based dynamic analysis can vet runtime behaviors of applications on a large scale. As a result, researchers have launched such systems [3, 4, 6, 23] to inspect file, network, and cellphone operations. Moreover, Google Bouncer [17] vets applications using QEMU-based emulators in its cloud infrastructure [20]. And Bouncer helped drop the number of malware downloads in Google Play by 40% in 2011 [17].

Despite the apparent success, a flaw of emulator-based dynamic analysis lies in the discrepancies between emulators and real devices. Such discrepancies, if observable by applications, may lead to detection heuristics (*a.k.a.*, "red pills") that indicate the fabricated reality of Android emulators. Taking advantage of these heuristics, Android malware can build split personalities and circumvent dynamic analysis as previously observed in PC malware [8]. Indeed, the security community has already discovered Android malware samples that use such heuristics to evade dynamic analysis [9, 11].

The heuristics in newly discovered Android malware samples, unlike their ancestors in PC malware, exploit the peculiarities of Android. Such heuristics check the presence or contents of certain artifacts (*e.g.*, files, APIs) in Android emulators. They do not depend on native code because native code has been of particular interest to some malware analysis systems [30]. Furthermore, they can be integrated into Android applications with simple algorithms. For example, the presence of a file `/sys/qemu_trace` indicates QEMU-based emulators [19]. A full-zero string returned by the Android API `getDeviceId` indicates Android SDK emulators [21]. Meanwhile, researchers have demonstrated the effectiveness of similar heuristics [21, 25]. This alarming trend calls for a comprehensive study of these detection heuristics to better understand their magnitude and accuracy.

ACSAC '14, December 08 - 12, 2014, New Orleans, LA, USA

http://dx.doi.org/10.1145/2664243.2664250.

Regrettably, all known detection heuristics that target Android emulators are discovered piece by piece in an ad-hoc fashion. For example, some heuristics are discovered through dissecting malware samples [9, 11]. Such a reactive approach cannot predict unknown heuristics. Other known heuristics are derived from manual analysis on specific components of Android emulators [19, 24, 25]. Even though this approach is proactive, manual analysis inevitably cannot address the multitude of components in Android emulators.

To convey the severity of the problem and get ahead of malicious adversaries, we propose a framework called Morpheus that proactively and automatically generates Android emulator detection heuristics. Unlike existing approaches, Morpheus retrieves and analyzes Android system artifacts observable by Android applications in their sandboxes, called *observable artifacts*. In particular, Morpheus starts from an analysis of the current Android application sandbox to identify sources of observable artifacts. Afterwards, Morpheus employs a probe application to automatically retrieve observable artifacts from both Android emulators and real devices. Morpheus then analyzes the artifacts and their contents to produce candidate heuristics. Finally, Morpheus ranks the candidates as its output.

We evaluate Morpheus with two steps. First, we apply Morpheus against widely deployed QEMU-based Android emulators and emerging VirtualBox-based emulators. The results are more than 10,000 novel detection heuristics that can be used to detect both types of emulators as a whole or either of them. We also investigate them to reveal and characterize the discrepancies between Android emulators and real devices. Second, we assemble a group of the top-ranked detection heuristics and evaluate their accuracies against 9 emulator-based Android malware analysis tools and 128 real devices. This group of heuristics could accurately detect the evaluated emulators and real devices. To better mitigate threats of emulator detection, we will release our discovered heuristics to the security community at `http://honeynet.asu.edu/morpheus/`[1].

We summarize the contributions of this paper as follows:

- *New techniques.* We develop new techniques that make the first step towards proactive and automated generation of Android emulator detection heuristics. We integrate these techniques into a framework, called Morpheus, to systematically generate heuristics that offer low false-positives and low false-negatives.
- *New findings.* We discover a large number of novel detection heuristics and reveal the underlying discrepancies between Android emulators and real devices. Our experiments against existing malware analysis tools and a large number of real devices demonstrate high accuracy of our discovered heuristics.

The remainder of this paper proceeds as follows. Section 2 provides the background of emulator detection and the threat model for our work. Section 3 describes the design of Morpheus and its components. Section 4 elaborates our discovered heuristics and their exploited discrepancies. Section 5 presents our experiments with a group of the top-ranked detection heuristics. Section 6 discusses countermeasures and limitations of our approach. Section 7 describes the related work. Section 8 concludes this paper.

[1]To prevent misuse, we may require verifying user identities before the dataset can be downloaded. Please visit our project website for further information.

2. BACKGROUND AND THREAT MODEL

In this section, we first describe the background of emulator detection in Android malware. We then present the attack model that this work is based on.

2.1 Detection Heuristics

Due to the peculiarities of Android, we argue that Android malware would be reluctant to reuse previous PC emulator detection heuristics. First, Android malware faces a unified runtime environment whose underlying implementation details (*e.g.*, hardware differences) are concealed by the Android middleware and APIs. At the same time, Android malware has been deprived of many capabilities that allow accessing low-level system artifacts by the Android application sandbox. In addition, Android malware would prefer detection heuristics implemented with Java code rather than native code. As native code is used by only a small fraction of benign Android applications but most malicious root exploits [30], native code would draw attention of analysis tools, breaking a detection heuristic's basic purpose of evading analysis.

The detection heuristics found in newly discovered malware samples seem to be in line with our argument. They allow an application to detect emulators without bypassing the application sandbox and without the assistance of native code. For example, a popular detection heuristic involves an Android API `getDeviceId` that returns the IMEI of an Android device. This heuristic calls `getDeviceId` and tests whether "`000000000000000`" is a substring of the returned value of `getDeviceId`. It can be implemented with only two lines of Java code and thus leaves relatively small footprints. Despite that researchers have discovered similar detection heuristics and evaluated their effectiveness against Android SDK emulators, the magnitude and accuracy of such heuristics remain unknown, which results in an impediment to the development of comprehensive countermeasures.

2.2 Threat Model

In our threat model, we assume emulators that run Android with default configurations. We also assume the presence of passive anti-detection techniques, which do not proactively instrument the application to suppress the execution of detection heuristics. This is also the common setup of the existing deployed emulator-based dynamic analysis systems.

In addition, we assume a malicious Android application that does not bypass the application sandbox or carry any native code. Meanwhile, we allow this application to request any Android permission. In other words, this application's capabilities are no more than those of the benign applications in application stores. Afterwards, it applies detection heuristics that check the presence or contents of certain artifacts. Based on the result, it determines whether it is running in an emulator or not.

Once this application detects emulators, it could stay dormant or exhibit legitimate behaviors. Furthermore, this application can use dynamic external code loading to evade both static and dynamic analysis, because it only downloads the malicious payload when it is in a real device. Alternatively, it can perform reconnaissance within the emulator and phone home to facilitate generation of up-to-date detection heuristics for future attacks. We attempt to understand whether the detection heuristics can be successful in terms of detecting Android emulators and real devices.

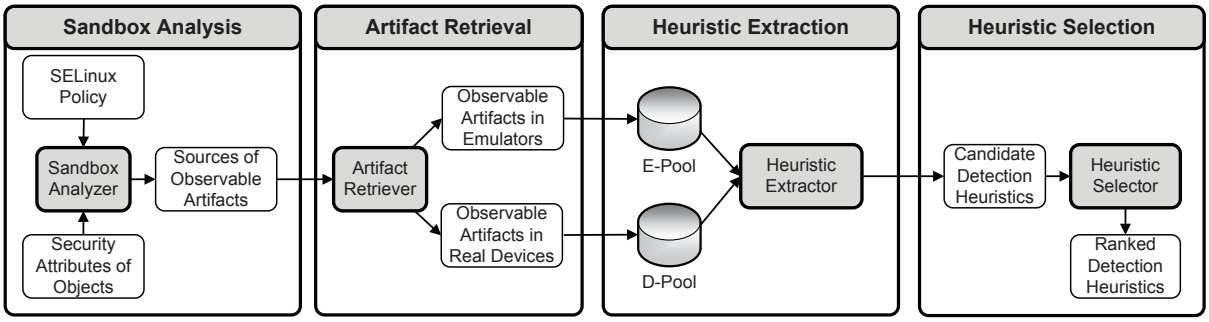


Figure 1: Morpheus: System Architecture

3. DESIGN AND SYSTEM COMPONENTS

In this section, we describe our framework Morpheus which is equipped with a suite of techniques for supporting automatic generation of detection heuristics. Despite the fact that Android emulators may contain plenty of discrepancies, a malicious application cannot exploit them unless it can observe them from within the application sandbox. Here we define *observable artifacts* as artifacts (*e.g.*, files, APIs) whose presence can be probed or whose contents can be read by any Android application in its sandbox. For example, suppose a file is not readable but its parent directory is listable, this file is still an observable artifact. The key idea of Morpheus is to retrieve and analyze observable artifacts.

As depicted in Figure 1, Morpheus consists of four components. The sandbox analyzer analyzes the default configurations of the Android application sandbox to identify sources of observable artifacts. For respective sources, the artifact retriever enumerates observable artifacts and retrieves their contents. The retrieved observable artifacts are uploaded to two pools for both emulators and real devices, respectively. The heuristic extractor then analyzes the pools by finding the artifacts or substrings of their contents that appear in most emulators but a small fraction of real devices, and vice versa. These artifacts and substrings constitute candidate detection heuristics. Finally, the heuristic selector ranks the candidates as the output of Morpheus. We next describe each component in detail.

3.1 Sandbox Analyzer

Applications' accesses on artifacts are regulated by the Android application sandbox, which is based on discretionary and mandatory access control (DAC and MAC). The Linux kernel provides DAC, which grants accesses by checking permissions of objects. Security-Enhanced Linux (SELinux) adds MAC over DAC starting from Android 4.3. SELinux grants accesses by checking domains of subjects (*e.g.*, `untrusted_app`), types of objects (*e.g.*, `wallpaper_files`), and SELinux permissions (*e.g.*, `open`, `read`)[2].

To identify sources of observable artifacts, we need to access all the objects in the Android OS. However, it is infeasible to do so in off-the-shelf Android devices due to the application sandbox and lack of root privileges. Instead, we propose the sandbox analyzer that analyzes the reference SELinux policy in Android and the security attributes (*e.g.*, owners, permissions, xattr) of objects in rooted reference Android devices (*e.g.*, Nexus devices). Specifically, we attempt to identify the objects whose security attributes expose themselves to third-party applications. Given that third-party applications are automatically assigned into the `untrusted_app` domain during installation, we simulate DAC and MAC checks to identify the following objects: (1) objects that are world-readable or under world-listable directories; and (2) objects that are accessible by `untrusted_app` using read-like SELinux permissions (*e.g.*, `read`, `recv_msg`, `ioctl`). From such objects, we then distill the sources of observable artifacts based on their owners and SELinux types, along with proper methods to retrieve them. For example, `/dev/binder` has the SELinux type `binder_device`. Its SELinux type indicates that it belongs to the Binder IPC subsystem that allows an application to access remote artifacts in system services. Such artifacts would require Binder-specific methods to retrieve. As variations in the hierarchy of objects across different Android versions are insignificant, the sources of observable artifacts derived from the reference inputs should be applicable in emulators and real devices.

We stress that the sandbox analyzer is much more conservative compared with the current Android application sandbox. SELinux in Android 4.3 is configured to permit every access. Even in Android 4.4, SELinux only protects several critical system daemons and does not confine third-party applications (*i.e.*, `untrusted_apps`). With that said, the true amount of observable artifacts in current Android devices could be much larger. However, considering the possibility that SELinux may extend its coverage in the upcoming versions of Android, we choose to be conservative for the future effectiveness of our detection heuristics.

3.2 Artifact Retriever

The artifact retriever is essentially a probe application. It requests all the available Android permissions to maximize its capabilities within the confinement of the application sandbox. Based on the identified sources of observable artifacts, we implement the corresponding methods in the artifact retriever to automatically retrieve the observable artifacts as well as their contents.

To address the various sources of observable artifacts, we propose three foundation modules in the artifact retriever: a directory walker, a Java function caller, and a Binder IPC caller. They are tailored to the peculiarity of Android and can be easily adapted and combined. Specifically, the directory walker traverses file-like artifacts. The Java function caller enumerates and manipulates both public and hidden Android APIs. The Binder IPC caller directly triggers remote system services (*e.g.*, `TelephonyManagerService`) with dynamically constructed Binder IPC messages.

[2]We ignore users, roles, and security levels for brevity because they are rarely used in the context of Android.

We launch the artifact retriever into both Android emulators and real devices. It probes the surrounding observable artifacts with its carried modules. It technically captures the first 1KB of each artifact's contents if readable. Upon explicit errors (*e.g.*, denied access), it records the error messages as the retrieved contents. Upon implicit errors (*e.g.*, blocking read), it uses a timeout to ensure that it does not hang there infinitely. We note that, the artifact retriever must upload artifacts to the correct pool according to where the artifacts are observed. For example, artifacts collected from emulators should never go into D-Pool. This is critical for the heuristic extractor to work effectively, because arbitrary noises could make the problem of heuristic generation NP-hard [16].

3.3 Heuristic Extractor

The inputs of the heuristic extractor are two pools, namely E-Pool and D-Pool, which contain instances of observed emulators and real devices, respectively. Each instance is a collection of key-value pairs that map retrieved artifacts to their contents. A key (artifact) occurs in an instance once at most, although it can occur in multiple instances. And a value (content) can be null if the artifact retriever fails to read the contents. Next, we describe two categories of detection heuristics that generate decisions based on the artifacts and their contents, respectively.

3.3.1 Artifact-based Heuristics

We start from a category of heuristics that make decisions based on the presence of artifacts. First, we attempt to discover the artifacts that are exclusively used by emulators, such as emulator-specific hardware, software, and configurations. As we use their presence to imply emulators, we refer to them as Type E artifacts. Furthermore, we also look for the artifacts that appear in most real devices, which become our Type D artifacts.

We propose two metrics, $COV_E(a)$ and $COV_D(a)$ to denote the fractions of instances in E-Pool or D-Pool that contain artifact a, *i.e.*, $COV_E(a) = \frac{|E_a|}{|E|}$, and $COV_D(a) = \frac{|D_a|}{|D|}$. Intuitively, our heuristics should at least perform better than a 50/50 guess. Thus, we choose Type E and Type D artifacts from all the artifacts in both pools according to their values of $COV_E(a)$ and $COV_D(a)$ as follows:

- **Type E artifacts:** $COV_E > 50\%$, $COV_D < 50\%$
- **Type D artifacts:** $COV_E < 50\%$, $COV_D > 50\%$

3.3.2 Content-based Heuristics

However, there are plenty of artifacts that are prevalent in both emulators and real devices. For example, Android APIs would have both COV_E and COV_D larger than 50%. Inspired by Hamsa [16], we propose a category of detection heuristics whose decisions are based on tokens, where token is a contiguous byte subsequence in the contents of an artifact. Similar to what we introduce for artifact-based heuristics, we attempt to find Type E and Type D tokens.

Specifically, for an artifact a and its retrieved contents in E-Pool, we extract a set of tokens by computing common substrings among the contents. We then extract another set of tokens for D-Pool. Combining these two sets of tokens as a token set T, we compute $COV_E(a,t)$ and $COV_D(a,t)$, which are the fractions of instances in E-Pool and D-Pool whose contents of artifact a contain token t, *i.e.*, $COV_E(a,t) = \frac{|E_{a,t}|}{|E|}$ and $COV_D(t) = \frac{|D_{a,t}|}{|D|}$. Based on the values of $COV_E(a,t)$ and $COV_D(a,t)$, we select two type of tokens as our content-based heuristics as follows:

- **Type E tokens:** $COV_E > 50\%$, $COV_D < 50\%$
- **Type D tokens:** $COV_E < 50\%$, $COV_D > 50\%$

There are various algorithms that effectively compute common substrings. We opt for a suffix array in our heuristic extractor. Constructing a suffix array runs in $O(nlogn)$ time in worst case scenario and consumes $5n$ bytes of memory, where n is the total size of the contents of an artifact in a pool. Extracting tokens from the constructed suffix array can be implemented using a binary search. Furthermore, as we prefer longer tokens in the context of generating detection heuristics, we add one more step to prune tokens that are substrings of the other tokens as long as they share the same COV_E and COV_D.

The output of the heuristic extractor is a set of Type E and Type D heuristics. Each heuristic is represented as a 3-tuple $(artifact, token, type)$. *token* can be null for artifact-based heuristics. *type* implies the decision to be made once the observed artifact/token matches the artifact/token specified in the heuristic. The matched Type E heuristics indicate emulators and the unmatched ones indicate real devices. Conversely, Type D heuristics imply the opposite decision.

3.4 Heuristic Selector

We propose the heuristic selector to rank the candidate detection heuristics generated by the heuristic extractor. In general, we reduce the problem of ranking the candidates to the problem of feature selection in supervised learning. E-Pool and D-Pool comprise a training set consisted of instances that are correctly labeled with "emulator" or "real device." Furthermore, we have extracted a set of detection heuristics that can be considered as binary features. Now we need to select the relevant and non-redundant detection heuristics that would best classify future observations.

We propose to use a random forest [14], which is an ensemble learning method that leverages a multitude of decision trees for classification. Each individual decision tree covers a random subset of the features and is trained with a random subset of training samples. Afterwards, the random forest fits the training set by letting each decision tree predict its unseen samples and evaluate the errors. During this process, an *importance score* for each feature is measured based on how significant the error rate would change if the feature is removed from the decision trees.

We use this importance score as a metric to rank the candidate heuristics. On one hand, relevant heuristics that contribute much to classification naturally get higher importance scores. On the other hand, redundant heuristics that exploit the same artifact/token as other heuristics are assigned zero or lower importance scores. As such, the final output of the heuristic selector is a set of relevant and non-redundant detection heuristics as sorted by their importance scores derived from the random forest.

As the number of detection heuristics is much larger than the number of instances in the pools, the random forest may suffer from over-fitting, which overestimates the importance level of some heuristics. To suppress over-fitting, we choose to increase the number of decision trees in the random forest. As more trees are added, its tendency to over-fit generally decreases as no single feature can affect every decision tree.

4. FINDING DETECTION HEURISTICS

We ran our experiments with Morpheus against QEMU-based Android SDK emulators [1], VirtualBox-based Genymotion emulators [2], and real devices. In this section, we elaborate our experiments that lead to the findings of 10,632 detection heuristics. We then characterize the heuristics according to the underlying discrepancies that they exploit.

4.1 Experimental Setup and Findings

To understand the observable artifacts in the reference Android devices, we adopted an instance of the SDK emulator and a Galaxy Nexus phone that both run Android 4.4. We traversed their mounted file systems to obtain the security attributes of objects. We then acquired a copy of the default SELinux policy from the Android Open Source Project (AOSP). Using these as inputs, the sandbox analyzer identified 33 sources of observable artifacts. However, retrieving all of them requires plenty of domain-specific knowledge for tasks such as enumerating artifacts and constructing valid inputs. In this work, we only retrieved 3 sources that could possibly lead to discrepancies and cover a sufficient number of observable artifacts.

Procfs and Sysfs: Procfs and sysfs are both pseudo file systems that expose kernel objects to userspace programs. Specifically, procfs presents system information, such as loaded kernel modules, mounted filesystems, and network stacks. Sysfs exports hardware information such as connected block devices, buses, and power states. Our implementation of the artifact retriever traversed these two file systems mounted at `/proc` and `/sys`. In particular, we slightly adapted the directory walker to handle looped symbolic links that are prevalent in procfs and sysfs.

Android APIs: A large number public and hidden APIs are exposed by Android system services. For example, `TelephonyManagerService` exposes APIs that return unique device identifiers to applications. Actually, the APIs are implemented with the underlying Binder IPC framework, which handles the IPC between applications and system services through a Binder device node located at `/dev/binder`.

To probe APIs behind Binder, we implemented two approaches in the artifact retriever. We used the reflection-based Java function caller to enumerate and call APIs. We also adapted the Binder IPC caller to construct and send IPC messages to the remote system services. The returned Java objects and Binder IPC messages were converted into byte sequences as the retrieved artifacts' contents. For Java objects that are not of Java primitive types, we leveraged their `toString` method to acquire more information about them. In this paper, we are particularly interested in Android APIs that do not have any input parameters. According to [22], such APIs are more likely to be "sources that return non-constant values into application code." As a result, we covered approximately 15% of the 1,326 APIs exposed by Android system services.

Android System Properties: Similar to the Windows registry, Android includes a subsystem that centrally stores system configurations and status. This subsystem, usually dubbed as "property system," has been extensively used by Android system services. For example, a system property `ro.kernel.qemu` is read by the Android debugging bridge daemon (adbd) to determine the presence of emulators. System properties also cover meta information about the hardware, such as device models and manufacturers. Despite that SELinux in Android protects system properties, we inspected the implementation of the property system and found that the security check is only in the function `property_set()`, meaning that SELinux does not prevent reading system properties at all. Moreover, applications are allowed to read `/dev/__properties__`, which is the interface to system properties. Therefore, system properties are observable by every installed application. To retrieve system properties, we adapted the artifact retriever to call a binary executable located at `/system/bin/getprop`. It enumerates system properties so that the Java function caller can read the contents of each property. We note that this executable is only for the artifact retriever. It is not required by the detection heuristics that read system properties.

Table 1: Discovered Detection Heuristics

Pools	Detection Heuristics			
	File	API	Property	Total
D-Pool + E-Pool	2,121	81	82	2,284
D-Pool + E_Q-Pool	2,961	163	132	3,256
D-Pool + E_V-Pool	4,782	150	160	5,092
Total	9,864	394	374	10,632

Afterward, we ran the adapted artifact retriever against 16 instances of QEMU-based SDK emulators, 11 instances of VirtualBox-based Genymotion emulators, and 25 real devices. The SDK emulators covered three CPU architectures, namely ARM, x86, and MIPS. The Genymotion emulators covered x86, which is the only architecture they support. Both emulator types covered Android versions from 2.3 to 4.4. The real devices covered four manufacturers (Samsung, HTC, Motorola, and LGE), three ARM SoC families (Qualcomm Snapdragon, Texas Instruments OMAP, and Nvidia Tegra), and Android versions from 2.1 to 4.4. In particular, the real devices were borrowed from the participants we recruited through university mailing lists under the study protocol reviewed by our institution's IRB. Anecdotally, it took approximately 5-20 minutes for the artifact retriever to retrieve and upload the observable artifacts on each device.

The retrieved artifacts contributed to four pools for QEMU-based emulators (E_Q-Pool), VirtualBox-based emulators (E_V-Pool), all the emulators (E-Pool), and real devices (D-Pool). We then fed these pools to the heuristic extractor and the heuristic selector. The heuristic selector ranked the candidate heuristics with 10,000 decision trees and pruned the heuristics with zero importance scores. Table 1 shows a breakdown of the discovered 10,632 detection heuristics. In the remainder of this paper, we will respectively refer to these three categories of heuristics as *file* heuristics, *API* heuristics, and *property* heuristics, in the interest of brevity.

4.2 Characterizing Detection Heuristics

Next, we characterize the discovered heuristics based on the discrepancies they exploit. We first discuss the common detection heuristics that exploit the discrepancies shared by both QEMU-based and VirtualBox-based emulators. We then discuss the heuristics that leverage the QEMU-specific or VirtualBox-specific discrepancies, respectively. Our discussion does not aim to be exhaustive, instead we attempt to convey the scope of discrepancies in Android emulators. Considering that an attacker can possibly use this section as hints to craft detection heuristics, we suggest provisional but deployable countermeasures in Section 6.

4.2.1 Common Detection Heuristics

Network. These detection heuristics exploit the discrepancies in network interfaces, Netfilter modules, and kernel modules. For example, we found that all the emulators exclusively use `eth0`, whereas the real devices use `wlan0` and `rmnet`. The emulators also miss several IPv6-specific interfaces. In addition, the network interfaces in the emulators are not tetherable, because the emulators are missing the Remote Network Driver Interface (RNDIS) drivers that enable tethering. Netfilter is another source of discrepancies. The real devices include Netfilter modules for several network protocols that are rarely used in the context of mobile devices. Finally, Android introduces a kernel module to track data usage of installed applications. This module does not exist in the emulators.

Power management. This type of heuristics focuses on the power management subsystem. For example, the emulators lack the voltage and current regulators. The emulated CPU does not support frequency scaling. Another heuristic lies in the prevalence of multi-core CPUs in real devices. All the emulators only have a single core, whereas 75% of the real devices have at least two cores.

Audio. A handy feature of Android is headset detection, which allows the audio output to automatically switch between speakers and headsets. This feature is supported by GPIO/I2C buses. Notably, the emulators do not emulate these buses, while 95.6% of the real devices in our experiments have them. Furthermore, the differences in the implementation of audio subsystems between the emulators and real devices result in disparate audio drivers.

USB. Recently, USB On-The-Go (OTG) has been widely adopted in popular Android phones and tablets. It allows mobile devices to act as hosts and control USB peripherals. Intuitively, the mobile devices have to pre-install corresponding drivers of USB peripherals. As a result, we found that the real devices in our experiments carry drivers for Apple Magic Mouse, joysticks, and external displays. On the contrary, the emulators do not have such drivers and do not support USB OTG.

Radio. The software-emulated radio can lead to detection heuristics as well. For instance, the name of the baseband in all the emulators is "`unknown`." Moreover, the emulators use a default reference implementation of the radio interface layer (RIL), while the real devices typically use customized ones with different names. Similarly, the phone numbers, voicemail numbers, device serial numbers of the emulators are also constants and can be fingerprinted.

Software components and configurations. Despite that most of the discovered detection heuristics are related to hardware, we also identified several heuristics that exploit certain software components and their configurations. For example, the emulators use unique input methods and search interfaces. Regarding configurations, a prominent example is the key that signs the Android OS. The emulators use test keys while the real devices use release keys.

4.2.2 QEMU Detection Heuristics

QEMU. We found various observable artifacts that are part of QEMU. For example, we found a device node that accelerates the virtual graphics. In addition, there are several system properties set by QEMU and read by Android system services. An example is a property that stores the pixel density of virtual screens.

Goldfish virtual hardware. Most existing QEMU-based Android emulators are built upon a virtual hardware platform called "Goldfish." This platform introduces a set of virtual hardware for QEMU to run Android as its guest operating system. For instance, this set of virtual hardware includes a framebuffer, an audio device, and a battery. They are a must for QEMU-based emulators but never appear in real Android devices.

Bluetooth, NFC, and vibrator. The current QEMU-based emulators do not support these hardware. Their corresponding Android APIs return null if called from within the emulators. In particular, the driver of the vibrator is based on a Linux driver model called `timed_output`, which is also missing from the emulators.

4.2.3 VirtualBox Detection Heuristics

VirtualBox. Similar to QEMU-based emulators, we also found plenty of VirtualBox-specific artifacts. For example, we found 4 kernel modules that belong to VirtualBox Guest Additions. As stated in VirtualBox's documentation, these modules "optimize the guest operating system for better performance and usability." However, their presence also indicates VirtualBox.

PC hardware. As we have discussed, QEMU-based emulators lack support for some popular hardware, such as Bluetooth and NFC. On the contrary, VirtualBox-based emulators support many types of hardware that Android does not need. We found hundreds of artifacts that indicate PC hardware and obviously should not appear in mobile operating systems. For example, we found artifacts related to ACPI, CPU fans, thermal sensors, CD-ROM drives, AC97 audio codecs, and PCI Express.

5. MEASURING DETECTION HEURISTICS

As we have demonstrated the magnitude of the detection heuristic for Android emulators, we further measure their accuracies. To this end, we assembled a group of the top-ranked detection heuristics which are ranked by the heuristic selector. We then tested them against emulator-based malware analysis tools and real devices. In this section, we describe our experiments along with an empirical study on the average accuracies.

5.1 Experimental Setup

As the generated common detection heuristics were already ranked by the heuristic selector, we selected the top 10 heuristics out of the **F**ile, **A**PI, and **P**roperty detection heuristics, respectively. Table 2 lists the artifacts, tokens, and types of the 30 selected detection heuristics.

We created a synthetic application[3] to simulate the Android malware as we described in the threat model. Specifically, this application integrated the 30 heuristics with a heuristic matching engine based on Java's substring searching methods. It generated its decision using a majority vote among the 30 heuristics. In other words, an Android device is recognized as an emulator if more than half of the detection heuristics indicate so. Furthermore, it only needed four permissions: `READ_PHONE_STATE`, `ACCESS_NETWORK_STATE`, `ACCESS_WIFI_STATE`, and `INTERNET`, which are also frequently requested by benign applications in Google Play [29].

[3] `https://play.google.com/store/apps/details?id=edu.sefcom.devicetester` and `http://goo.gl/FXspKw`

Table 2: Top 10 File, API, and Property Heuristics

	Artifact	Token	Type
F1	/proc/misc	"network_throughput"	E
F2	/proc/ioports	"0ff\0:"	E
F3	/proc/uid_stat		D
F4	/sys/devices/virtual/misc/ cpu_dma_latency/uevent	"MINOR=5"	E
F5	/sys/devices/virtual/ppp		D
F6	/sys/devices/virtual/switch		D
F7	/sys/module/alarm/parameters		D
F8	/sys/devices/system/cpu/ cpu0/cpufreq		D
F9	/sys/devices/virtual/misc/ android_adb		D
F10	/proc/sys/net/ipv4/ tcp_syncookies		E
A1	isTetheringSupported()	"false"	E
A2	getAuthenticatorTypes()	"AuthenticatorDescription {type=com.g}"	D
A3	getSystemSharedLibraryNames()	"com.g"	D
A4	getGlobalSearchActivity()	".android.quicksearchbox/ com.android.quicksearchbox"	E
A5	getWebSearchActivity()	".android.quicksearchbox/ com.android.quicksearchbox"	E
A6	getTetherableWifiRegexs()	"wlan0\n"	D
A7	getTetherableUsbRegexs()	"rndis"	D
A8	getEnabledInputMethodList()	".android.inputmethod .latin/."	E
A9	getDeviceId() via Binder	"\0\0\03"	D
A10	getTetherableIfaces()	"wlan0"	D
P1	qemu.hw.mainkeys		E
P2	ro.build.description	"release-keys"	D
P3	ro.build.fingerprint	":user/release-keys"	D
P4	net.eth0.dns1		E
P5	rild.libpath	"/system/lib/libreference-ril.so"	E
P6	ro.radio.use-ppp		E
P7	gsm.version.baseband		D
P8	ro.build.tags	"release-key"	D
P9	ro.build.display.id	"test-"	E
P10	init.svc.console		E

Table 3: Evaluated Emulators and Real Devices

Emulators (9)	DroidBox [5] 2.3 and 4.1, Andrubis [6], CopperDroid [23], SandDroid [3], TraceDroid [4], Qihu 360, NVISO ApkScan, ForeSafe
Real Devices (128)	Samsung, HTC, LGE, Huawei, Motorola, Sony Ericsson, Lenovo, ZTE, Hisense, Asus, Acer, OPPO, BBK, Meizu, Gionee, DOOV, YuLong, Haier, AMOI

We ran this application in 9 emulator-based malware analysis tools and 128 distinct real devices. As shown in Table 3, the malware analysis tools covered two versions of an offline tool called DroidBox and 7 online services. Among the online services, 4 are derived from previous research work and 3 are security products. The 128 real devices were from AppThwack, TestObject, and Baidu MTC, all of which are online services that automatically test applications in real phones and tablets. Note that we did not run our artifact retriever on them due to their limited device minutes and bandwidth quota.

5.2 Results and Empirical Analysis

We deem emulators as *positive* and real devices as *negative*. Given the measured true positives (TP), false negatives (FN), false positives (FP), and true negatives (TN), we attempt to evaluate the detection heuristics with three metrics, namely *sensitivity*, *specificity*, and *accuracy*. For example, a Type E detection heuristic is sensitive if it matches all the emulator instances. And, it is specific if it does not match any non-emulator instances, *i.e.*, real devices. Simply put, we compute the values of the three metrics as follows:

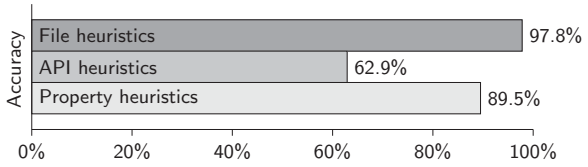
- $Sensitivity = TP/(TP + FN)$;
- $Specificity = TN/(FP + TN)$; and
- $Accuracy = (TP + TN)/(TP + FN + FP + TN)$.

Table 4: Evaluation Results of the 30 Heuristics

	TP	FN	FP	TN	Sens.(%)	Spec.(%)	Acc.(%)
F1	9	0	2	126	100.0	98.4	98.5
F2	9	0	0	128	100.0	100.0	100.0
F3	9	0	7	121	100.0	94.5	94.9
F4	9	0	4	124	100.0	96.9	97.1
F5	9	0	1	127	100.0	99.2	99.3
F6	9	0	1	127	100.0	99.2	99.3
F7	9	0	7	121	100.0	94.5	94.9
F8	9	0	0	128	100.0	100.0	100.0
F9	9	0	0	128	100.0	100.0	100.0
F10	9	0	8	120	100.0	93.8	94.2
A1	9	0	3	125	100.0	97.7	97.8
A2	7	2	82	46	77.8	35.9	38.7
A3	5	4	24	104	55.6	81.3	79.6
A4	7	2	48	80	77.8	62.5	63.5
A5	7	2	45	83	77.8	64.8	65.7
A6	9	0	37	91	100.0	71.1	73.0
A7	9	0	64	64	100.0	50.0	53.3
A8	9	0	37	91	90.0	71.1	72.5
A9	6	3	72	56	66.7	43.8	45.3
A10	9	0	82	46	100.0	35.9	40.1
P1	8	1	2	126	88.9	98.4	97.8
P2	8	1	17	111	88.9	86.7	86.9
P3	8	1	21	107	88.9	83.6	83.9
P4	9	0	5	123	100.0	96.1	96.4
P5	9	0	2	126	100.0	98.4	98.5
P6	9	0	11	117	100.0	91.4	92.0
P7	9	0	14	114	100.0	89.1	89.8
P8	8	1	10	118	88.9	92.2	92.0
P9	8	1	0	128	88.9	100.0	99.3
P10	9	0	20	108	100.0	84.4	85.4

Table 4 demonstrates the measured accuracies of the 30 detection heuristics. We next present our empirical analysis on the average accuracies from three aspects.

5.2.1 File, API, and Property Heuristics



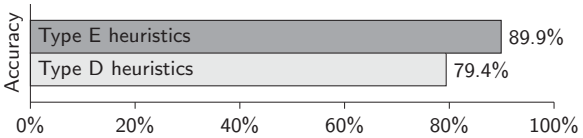
We first inspected the average accuracies of the heuristics according to the categories of their exploited observable artifacts. As shown in the above bar chart and Table 4, the file heuristics enjoyed both high sensitivities and specificities with an average accuracy of 97.8%. The API heuristics, despite of their acceptable sensitivities, suffered from significantly low specificities. For example, A2, A9, and A10 performed no better than 50/50 guesses as their accuracies were less than 50%. The property heuristics performed fairly good with an average accuracy of 89.5%.

One possible explanation for the API heuristics' low accuracies is that the Android APIs are designed to provide some sort of hardware/software abstraction. An evidence is the Android Compatibility Program[4], which precisely defines the behaviors of Android APIs to ensure that Android applications run in "a consistent and standard environment."

[4]`https://source.android.com/compatibility`

To build such an environment, the APIs that reveal the underlying details are not necessary, and they are subject to be removed or deprecated. However, we argue that this environment also requires a well-configured application sandbox to prevent applications from bypassing the APIs. Unfortunately, our discovered file and property heuristics imply that the current sandbox should be reinforced.

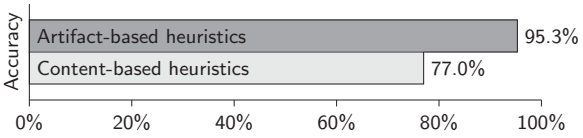
5.2.2 Type E and Type D Heuristics



We investigated the differences between the Type E and Type D heuristics. As we have discussed in Section 3.3, Type E and Type D detection heuristics respectively indicate emulators and real devices. In our experiments, the Type E heuristics outperformed the Type D ones.

We note that almost all of the heuristics in Table 4 with low specificities are of Type D. We believe that it is due to the diversified and fragmented nature of real devices. Type D heuristics expect the artifacts/tokens that are prevalent in real devices. However, device manufacturers inevitably customize devices and change artifacts, which makes it harder to find the artifacts that exist in every real device. On the contrary, emulators are much more unified in terms of customizations, which is possibly due to the difficulty in modifying and maintaining software-emulated hardware.

5.2.3 Artifact-based and Content-based Heuristics



Finally, we compared the artifact-based and content-based heuristics. The bar chart shows that the artifact-based heuristics had an average accuracy of 95.3%. The content-based heuristics fell behind with 77.0%. However, we stress that F1, F4, and P9 are also content-based heuristics and their accuracies were among the top of the 30 heuristics.

In addition to the factors of abstraction and customization that we discussed earlier, a possible explanation is that content-based heuristics are more subtle and vulnerable to intended or unintended changes. Content-based heuristics exploit artifacts' contents (*e.g.*, configurations), which are subject to change in a rapidly evolving system like Android. On the contrary, artifact-based heuristics rely on the presence of certain artifacts. Compared with the contents, the artifacts (*e.g.*, kernel modules) are relatively consistent in emulators and real devices, because developers are usually reluctant to remove them as to avoid unexpected problems.

5.3 Case Study: A9

Finally, we present a case study on heuristic A9 because it involves an Android API `getDeviceId`, which has been popular among the known detection heuristics. A9 exploits the same API but in a slightly different way. Specifically, it looks for a token "`\0\0\03`" in the Binder IPC message returned from the implementation of `getDeviceId` in `TelephonyManagerService`. In other words, A9 uses the IMEIs whose first character is "`3`" to indicate real devices. However, it turned out that A9 had a sensitivity of 66.7% and an unbearable specificity of 43.8%. We investigated the evaluated emulators and real devices to find out how they reacted to A9. The investigation led us to flaws in an existing anti-detection technique and improvements for A9.

5.3.1 A9 vs DroidBox 4.1

As we discussed in Section 4.1, our implementation of the artifact retriever employs a Java function caller and a Binder IPC caller to probe Android APIs. When we tested A9 against DroidBox 4.1, we found that these two callers returned disparate values, namely "`357242043237511`" and "`000000000000000`." We note that both callers should return the same value, because the application-side Binder proxies of Android APIs are not supposed to modify the IPC messages returned by the underlying Binder stubs in system services.

We inspected DroidBox's source code and found that DroidBox rewrites the Binder proxy of `getDeviceId` to return a dummy IMEI without involving the Binder stubs. Although such a countermeasure could neutralize the detection heuristics that call `getDeviceId` in a normal way, it is not effective against the Binder IPC caller, which bypasses the countermeasure and reads the actual full-zero IMEI. Therefore, we believe that the countermeasure implemented in DroidBox 4.1 is not complete. We note that ApkScan demonstrated the same issue, implying that ApkScan might have integrated DroidBox 4.1 for its dynamic analysis.

However, A9 was not effective against DroidBox 2.3. We found that DroidBox 2.3 opts for a similar countermeasure but implements it in the service-side Binder stub. In such a case, bypassing the stub and observing the actual IMEI would require root privileges, *i.e.*, the actual IMEI is not observable. Therefore, such a countermeasure is effective and the dummy IMEI would appear realistic.

5.3.2 A9 vs Non-U.S. Devices

A9 assumes that an Android device whose IMEI starts with "`3`" is a real device, otherwise it is an emulator. We checked the IMEIs of the 128 real devices and found that this assumption is incorrect.

According to IMEI Allocation and Approval Guidelines [7], the first digit of an IMEI is part of Reporting Body Identifier (RBI), which identifies the GSMA-approved authority that issues the IMEI. Typically, IMEIs of mobile devices are issued by the authorities in the same area where the devices are sold. For example, IMEIs of the devices sold in the U.S. are issued by the British Approvals Board for Telecommunications (BABT) and thus start with BABT's code "`35`." Similarly, IMEIs of the devices sold in China start with "`86`." We note that about half of the 128 evaluated real devices were from Baidu MTC that uses Android phones sold in China. Given that A9 was based on the devices in the U.S., A9 naturally got a low specificity, and it could be improved with wild cards that match multiple RBIs.

The lesson of A9 indeed illustrates the future of the armed race between emulator detection and anti-detection. First, Android malware could check the semantics of the observed artifacts. For example, the dummy IMEI in DroidBox 4.1 is invalid and could be noticed by a sophisticated adversary. Second, emulator-based malware analysis tools should consider the observability of actual artifacts and the semantics of dummy artifacts to be less distinguishable.

6. DISCUSSION

The evaluation results imply an imminent threat that Android malware may thwart existing emulator-based dynamic analysis systems. In this section, we suggest the potential countermeasures and discuss the limitations of our work.

6.1 Countermeasures

Provisional countermeasures. We suggest the methods that detect the usage of detection heuristics in Android malware as provisional countermeasures. Although they do not prevent Android malware from detecting Android emulators, they can raise alarms for analysts and thus thwart the malware's original purpose of evading analysis. For example, dynamic analysis systems could monitor accesses on files and properties seldom used by benign applications. API heuristics are much more stealthy because benign applications also frequently use them. In such a case, we suggest static data-flow analysis to locate branches that involve detection heuristics and lead to disparate code blocks.

Short-term countermeasures. Next, we discuss the countermeasures that allow an emulator to appear "realistic" to Android malware. First, we suggest a comprehensive deployment of dummy artifacts. Some existing works can be adapted to facilitate such countermeasures. For example, AirBag [26] supports a decoupled and isolated runtime environment based on OS-level virtualization. ASM [13] provides programmable interfaces that interpose Android APIs and return dummy values to applications. These works, if combined and extended, can enable a "brain in a vat" setup where an application runs in an emulator but gets dummy and valid data originated from real devices. Second, we suggest denying accesses on unnecessary observable artifacts with strict DAC and MAC policies. For example, artifacts in sysfs exploited by our file heuristics seem unnecessary for general Android applications. However, the usability impact of denying accesses still needs further verification.

Long-term countermeasures. The ideal countermeasure is to fix all the discrepancies in Android emulators. Although Garfinkel *et al.* [12] concludes its infeasibility in 2007 due to the inherent hardness of creating indistinguishable software-emulated hardware, hardware-assisted virtualization techniques (*e.g.*, Intel VT-x and VT-d) have evolved significantly to allow PC emulators to virtualize real hardware. Currently, ARM CPUs have integrated necessary virtualization extensions. Meanwhile, commodity ARM hypervisors are also in active development. We envision emerging Android emulators equipped with virtualized CPUs, sensors, and radios in the near future.

6.2 Limitations

Despite the robustness of Morpheus, the quality of the discovered detection heuristics is limited by the small number of real devices used in finding detection heuristics (Section 4). In general, Morpheus works like supervised learning, and its performance inevitably depends on the quality of the "training set," *i.e.*, the emulators and real devices observed by the artifact retriever. We note that the artifact retriever needs approximately 20 minutes to collect the artifacts on a single device. Unfortunately, online services like AppThwack (Section 5) do not allow the artifact retriever to run for such a long time or upload large bulks of data. As for future work, we plan to reach out to mobile carriers and device vendors to collect observable artifacts from more real devices.

Although Morpheus discovered more than 10,000 heuristics, we stress that they were derived only from 3 out of 33 sources of observable artifacts. To better understand the scope of detection heuristics for effective countermeasures, the artifact retriever could be enhanced to address more sources of artifacts as well as sophisticated usages of them. Examples include extended modules of the artifact retriever that can handle callbacks or construct valid input parameters for Android APIs. We did not cover them in this work because they require domain-specific knowledge of each Android system service.

Our heuristic generator produces relatively rigid heuristics, such as A9 that does not match multiple RBIs. This can be improved with more sophisticated and flexible heuristics. For example, a token-sequence heuristic matches an ordered set of tokens in the contents of an artifact. Moreover, a naïve Bayes heuristic enables probabilistic matching by aggregating the empirical probabilities of multiple artifact/token heuristics with the Bayes' law, assuming that the occurrences of artifacts/tokens are independent.

7. RELATED WORK

Behavior-based detection heuristics. Researchers have proposed several heuristics that exploit discrepancies in runtime behaviors rather than artifacts. For instance, a piece of specially crafted native code can identify QEMU-based emulators due to the discrepancies in QEMU's caching behaviors [19, 21, 24]. Low video frame rate indicates emulators because of the performance drawbacks in the SDK emulator's graphics rendering engine [25]. However, these heuristics are not evaluated against VirtualBox-based emulators and real devices. Thus, their sensitivities and specificities require further investigation. In addition, these heuristics do not return a decision until a sufficient number of events are observed, which tends to increase their footprints and attract analysis. Along these lines, Morpheus addresses artifact-based and content-based detection heuristics. More importantly, Morpheus generates detection heuristics automatically and systematically.

Dynamic analysis frameworks. Researchers have built several dynamic analysis frameworks to vet the runtime behaviors of Android malware. TaintDroid [10] tracks information flows that leak sensitive data to the Internet. VetDroid [28] further reveals information flows that involve permissions. AppIntent [27] helps determine if an information flow is user-intended. Some of these tools have been integrated into automated malware analysis systems such as DroidBox [5], Andrubis [6], CopperDroid [23], SandDroid [3], and TraceDroid [4]. They are vulnerable to be evaded using the detection heuristics in this work as long as they are deployed in Android emulators.

8. CONCLUSION

Recent Android malware demonstrates the capabilities of detecting Android emulators using detection heuristics. To convey the severity of this problem, we have presented Morpheus, a system that automatically and systematically generates detection heuristics. Morpheus analyzes artifacts observable by Android applications and discovers exploitable discrepancies in Android emulators. Moreover, we have described a proof-of-concept implementation of Morpheus, along with extensive experiments and findings.

Acknowledgements

We would like to thank Adam Doupé and the anonymous reviewers for their valuable comments that helped improve the presentation of this paper. This work was supported in part by the National Science Foundation and National Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of the funding agencies.

9. REFERENCES

[1] Android developers - using the emulator. http://developer.android.com/tools/devices/emulator.html. Accessed: May 2014.

[2] Genymotion, the fastest android emulator for app testing and presentation. http://genymotion.com. Accessed: May 2014.

[3] Sanddroid - an apk analysis sandbox. http://sanddroid.xjtu.edu.cn/. Accessed: May 2014.

[4] Tracedroid - dynamic android app analysis (by vu amsterdam). http://tracedroid.few.vu.nl/. Accessed: May 2014.

[5] Droidbox: An android application sandbox for dynamic analysis. https://code.google.com/p/droidbox/, 2011. Accessed: May 2014.

[6] Andrubis: A tool for analyzing unknown android applications. http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/, June 2012. Accessed: May 2014.

[7] G. Association et al. Imei allocation and approval guidelines. volume 10, 2010.

[8] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient detection of split personalities in malware. In *Proceedings of Network and Distributed System Security Symposium*, 2010.

[9] H. Dharmdasani. Android.hehe: Malware now disconnects phone calls. http://www.fireeye.com/blog/technical/2014/01/android-hehe-malware-now-disconnects-phone-calls.html, January 2014. Accessed: May 2014.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX conference on Operating systems design and implementation*, pages 1–6. USENIX, 2010.

[11] F-Secure. Trojan:android/pincer.a. http://www.f-secure.com/weblog/archives/00002538.html, April 2013. Accessed: May 2014.

[12] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: Vmm detection myths and realities. In *Proceedings of USENIX Workshop on Hot Topics in Operating Systems*, 2007.

[13] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. Asm: A programmable interface for extending android security. In *Proceedings of the USENIX Security Symposium*, 2014.

[14] T. K. Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.

[15] C. Ionescu. Obfuscating embedded malware on android. http://www.symantec.com/connect/blogs/obfuscating-embedded-malware-android, June 2012. Accessed: May 2014.

[16] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 15–pp. IEEE, 2006.

[17] H. Lockheimer. Android and security. http://googlemobile.blogspot.com/2012/02/android-and-security.html, February 2012. Accessed: May 2014.

[18] I. Lunden. Gartner: 102b app store downloads globally in 2013, 26b in sales, 17% from in-app purchases. http://techcrunch.com/2013/09/19/gartner-102b-app-store-downloads-globally-in-2013-26b-in-sales-17-from-in-app-purchases/, September 2013. Accessed: May 2014.

[19] F. Matenaar and P. Schulz. Detecting android sandboxes. http://dexlabs.org/blog/btdetect, August 2012. Accessed: May 2014.

[20] J. Oberheide and C. Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[21] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the European Workshop on System Security*, page 5. ACM, 2014.

[22] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[23] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the European Workshop on System Security*, April 2013.

[24] P. Schulz. Android emulator detection by observing low-level caching behavior. https://bluebox.com/technical/android-emulator-detection-by-observing-low-level-caching-behavior/, December 2013. Accessed: May 2014.

[25] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*. ACM, 2014.

[26] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. Airbag: Boosting smartphone resistance to malware infection. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[27] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the ACM conference on Computer and communications security*, pages 1043–1054. ACM.

[28] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the ACM conference on Computer and communications security*, pages 611–622. ACM, 2013.

[29] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 95–109. IEEE, 2012.

[30] 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 *Proceedings of the Network and Distributed System Security Symposium*, pages 5–8, 2012.