Morpheus: Benchmarking Computational Diversity in Mobile Malware

Mikhail Kazdagli
University of Texas at Austin
mikhail.kazdagli@utexas.edu

Ling Huang
Intel Labs
ling.huang@intel.com

Vijay Reddi
University of Texas at Austin
vj@ece.utexas.edu

Mohit Tiwari
University of Texas at Austin
tiwari@austin.utexas.edu

ABSTRACT

Computational characteristics of a program can potentially be used to identify malicious programs from benign ones. However, systematically evaluating malware detection techniques, especially when malware samples are hard to run correctly and can adapt their computational characteristics, is a hard problem.

We introduce Morpheus – a benchmarking tool that includes both real mobile malware and a synthetic malware generator that can be configured to generate a computationally diverse malware sample-set – as a tool to evaluate computational signatures based malware detection. Morpheus also includes a set of computationally diverse benign applications that can be used to repackage malware into, along with a recorded trace of over 1 hour long realistic human usage for each app that can be used to replay both benign and malicious executions.

The current Morpheus prototype targets Android applications and malware samples. Using Morpheus, we quantify the computational diversity in malware behavior and expose opportunities for dynamic analyses that can detect mobile malware. Specifically, the use of obfuscation and encryption to thwart static analyses causes the malicious execution to be *more* distinctive – a potential opportunity for detection. We also present potential challenges, specifically, minimizing false positives that can arise due to diversity of benign executions.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software

Keywords

security, mobile malware, performance counters

1. INTRODUCTION

Mobile applications turn sensitive information – financial, business, location, health, among others – into compelling functionality for users. Unfortunately, the mobile ecosystem draws hordes of malware developers who exploit sensitive data and paid services for profit. Kaspersky, an IT security company, detected 100,000 new malicious programs in 2013, up from 40,059 in 2012 [10]. Further, Kaspersky reports finding approximately 4M benign "apps" embedded with malicious payloads to generate more than 10M malicious applications in 2012-2013. Desktop anti-viruses are ineffective and energy inefficient for mobile platforms, and detecting and eliminating malware on mobile devices remains a major problem.

We consider the potential use of a program's computational characteristics to detect malware. The system we consider has two synergistic components. On the server side, a platform provider (e.g., Google) creates a database of computational signatures by executing applications and measuring hardware performance counters, and attaches the signature to each application in the app-store. On client devices, a light-weight system service samples performance counters to create run-time traces from applications, and compares them to the signature database either on the device or the server. Note that the signature database can contain either a blacklist (of malware executions [23]) or a whitelist (of benign executions). In case malware is detected, the system raises a signal to alert the user and/or the system. The server will further refine its compute signature database using traces uploaded by clients.

Evaluating computational characteristics based defenses against mobile malware is extremely challenging. One reason is that malware samples available online seldom work correctly – their command and control servers are not functional; they were targeted for an older, vulnerable platform; they run only in specific geographical areas; or that the malicious payload is triggered only as a response to specific user actions – and experiments with malware require care to establish that malware did execute in a realistic manner.

Further, malware developers can adapt their programs to proposed defenses. Malware is often repackaged into popular applications and choosing an appropriate baseline application can hide the malware's compute or networking activity better. Malware can even adapt its payload to change its signature and avoid detection.

In this paper, we present Morpheus – a tool to help conduct computational signature based research studies. Morpheus comprises of a malware benchmark suite that includes both real malware samples and a set of configurably diverse, synthetic malware samples. We first extend prior surveys of mobile malware to characterize malware behaviors. We then select a representative set of real malware, reverse-engineer them, and complete their server-side functionality. Finally, we construct a tool that allows a researcher to

HASP 2014 Minneapolis, MN USA

choose from a list of malicious behaviors, specify detailed parameters specific to each behavior, select obfuscation techniques including the baseline application to be repackaged into, and automatically synthesizes a set of computationally diverse, repackaged, obfuscated malware samples.

2. UNDERSTANDING MOBILE MALWARE

Mobile malware is qualitatively different from desktop variants. The same-origin policy and permissions architecture implemented in mobile platforms motivates malware to gain privileges by asking for and abusing permissions. Malware spreads primarily through repackaged apps in third party app-stores and through phishing scams, and covers behaviors from stealing information to conducting network fraud. In this section, we present key insights gained from studying existing malware samples that guide the design of Morpheus.

We analyzed 53 malware families from 2012 and 19 from 2013 – a total of 175 malware samples in 72 families – downloaded from public malware repositories [2, 8]. Our study extends prior work – the malware genome project [37] – that analyzed over 1000 malware samples to trace their families, origins, and behaviors.

Our goal is to identify knobs to change the *computational behavior* of malware and to determine concrete values for these knobs (e.g., amount and rate of data stolen). Malware samples from the NCSU dataset don't reliably execute on current Android machines, so we chose to fix a small set of representative malware samples and then construct malware that we can run reliably and adapt precisely.

To study malware behavior, we disassembled the binaries (APKs on Android) and executed them on both an Android development board and the Android emulator to monitor a) permissions requested by the application, b) middleware-level events (such as the launch of Intents and Services), c) system calls, d) network traffic, and e) descriptions of malware samples from the malware repositories. We describe our key inferences below.

Analyze payloads instead of exploits. We found that most mobile malware families achieve their end goals by getting users to install overprivileged applications [25, 37] instead of through root exploits [12, 5, 4]. We observed root exploits in 10/143 samples in 2012 and 3/32 samples in 2013. To become over-privileged, malware comes as payload that is repackaged into popular benign applications and then distributed through third-party app-stores. This corroborates the NCSU dataset [37], which found 86% of malware to be repackaged.

Malware behaviors. At a high level we assigned every malicious payload to one or more of three behaviors: *information stealers, networked nodes*, and *computational nodes*.

Information stealers look for sensitive data and upload it to the server. User-specific sensitive data includes contacts, SMSs, emails, photos, videos, and application-specific data such as browser history and usernames, among others. Device-specific sensitive data includes identifiers – IMEI, IMSI, ISDN – and hardware and network information. The volume of data ranges from photos and videos at the high end (stolen either from the SD card or recorded via a surveillance app) to SMSs and device IDs on the low end.

The second category of malicious apps requires compromised devices to act as nodes in a network (e.g., a botnet). Networked nodes can send SMSs to premium numbers and block the owner of the phone from receiving a payment confirmation. Malware can also download files such as other applications in order to raise the ranking of a particular malicious app. Click fraud apps click on a specific web links to optimize search engine results for a target.

We anticipate a new category of malware that uses mobile devices as compute nodes, specifically mobile counterparts of desktop malware that runs password crackers or bitcoin miners on compromised machines. Over last couple months, several malware samples mining crypto currencies have been detected. We expect more mobile miners come to the foreground as devices get multiple CPUs and more memory (DRAM and Flash). Hence, we use password crackers as a plausible choice for compute-oriented malware.

3. BASELINE: REVERSE ENGINEERING REAL MALWARE

In this section, we present experiments with real malware samples and discuss issues with running them in a controlled environment. Based on our malware classification, we include in Morpheus at least one representative sample from each malware category.

3.1 Infostealers

For our experiments we chose two popular families that pack several malicious features: Trojan-Spy.AndroidOS.Geinimi.a[1] and Backdoor.AndroidOS. Sinpon.a[2]. Neither of these samples run unmodified. The malware samples check the runtime environment used to execute Android apps (e.g., geographic location and other IDs) to detect whether a malware is executed within an emulator or in a geographic region where it is not supposed to work – implying that someone is trying to reverse engineer the malware-ridden application. If something suspicious is detected, the malicious payload is not executed and the underlying application operates normally.

Geinimi. We found a Geinimi.a sample embedded within a repackaged version of the computationally intensive game MonkeyJump2. Geinimi.a is a complex and heavily obfuscated malware family. Besides employing obfuscation it also encrypts almost all strings and uses Java reflection for dynamic method invocation – all features to impede reverse-engineering of Geinimi.a. The Geinimi.a malware mainly focuses on stealing personal information, but also posseses the capability to install more malicious apps (attempting root exploits or using phishing).

Analysis of the network traffic generated by the sample showed that its server Command&Control (C&C) is no longer functional. The Geinimi.a sample tried to connect to the C&C server every 5 min to check whether or not to issue any commands on the host system. Since we did not observe any replies from C&C server, we wrote our own C&C server and added instrumentation code to make sure that malicious payload is actually issued. All communication with Geinimi.a malware is encrypted using 52-DES with a hardcoded key 0x0102030405060708.

The interesting feature of Geinimi.a is that it splits all incoming commands (e.g. app installation, stealing contacts/sms, accessing phone IDs and etc) in one of the two groups. Semantically, the groups are almost identical, but their internal implementation is different. It is worth noting that some of them are not completely implemented. For example, the first implementation of app installation functionality works only on rooted devices, thus it is able to install an app without asking for the owner's approval. The other

[1]Geinimi MD5 e0106a0f1e687834ad3c91e599ace1be

[2]Sinpon MD5 d4a557ec086e52c443bde1b8ace51739

implementation shows a pop-up message asking a user to approve to approve installation of a new apk. We exercised 9 commands of one type; for 8 of these commands the difference in their implementation is negligible and thus insignificant for our experiments. The last command, downloading and installation of an app, is implemented differently across these two groups, but only one version works reliably.

When first running the app infected by Geinimi.a, when the malware would try to access the cellular network location service it would crash. The lack of this hardware component on the Arndale board and no hardware stub to spoof it caused the program to crash. We replaced access to network service with access to the GPS service.

The original code was able to extract all of the text messages, encrypt them, and write to an open Java socket, but not commit them. To fix this partial execution, we augmented the malware to commit text messages. When testing the 3rd party program installation functionality, we observed that the malware failed silently because it could not write to SD card (Android boots off the SDCard on the Arndale board and an extra partition on the card was not made to spoof it). We modified the code to store downloaded apk files in the /data folder and automatically set the required permissions instead. The above changes yielded a fully functional Geinimi.a sample.

Sinpon. The other infostealer we studied represents the Sinpon.a malware family. It is capable of executing 20 different commands from a C&C server and can send sensitive data back to the C&C server. As in the previous case, C&C server did not appear to be alive so we implemented our own C&C server that supported Sinpon.aâĂŹs protocol.

We found out that Sinpon.a does not run on Android 3.0 and above because since Android 3.0 Google has prohibited establishing network connection within the appâĂŹs main thread. Thus, we had to modify Dalvik byte code in order to move networking to a background thread and successfully execute Sinpon.a on Android 4.2.1. We embedded Sinpon.a in three apps - Angry Birds, Sana med. app and TuneInRadio - and ensured correct execution of all 20 commands – "getContactInfo", "getMessagein", "LoadFile/data/IMG.JPG", "getInstalledApp" etc. – that it can accept from its C&C server.

3.2 Networked node: Click Fraud

In this experiment we used the sample that is identified as AdWare.AndroidOS.FetGp.a[3] by Kaspersky antivirus. The malware pretends to be a Google Play app, but it does not do anything useful. When a user launches the app, it shows only a white screen, and at the same time it starts a background service that keeps running even after the app is closed. The background service registers the mobile device by contacting http://www.mobilefilmizle.com/ipzaman.php and this server checks the device's network configuration and if the deviceâĂŹs IP address has changed. The service then sets a timer to wakeup event every 2 minutes. When the application awakes, it contacts the other server to retrieve a set of keywords which are later entered into Google one-by-one. The malware opens the search results for each query and scrapes the result page for links containing a certain format to click them using JavaScript mouse events. This allows the malware to click on advertising links, generating money for its developers. When testing the sample, we made sure that its C&C server was alive by capturing network traffic and instrumenting Dalvik bytecode to spoof the command on subsequent calls.

3.3 Compute Node

Recently, we have found several mobile malware samples that are able to mine cryptocurrencies [9]. However, we did not observe computationally intensive malware in the wild before paper submission. Hence, we obtained a real password cracker[4] from Google Play store. The app uses brute-force search to break the MD5 hash of a password. We injected the code into legal versions of our benign application benchmark set.

4. SYNTHETIC MALWARE

Based on the analysis of a large set of Android malware samples, we developed a highly customizable malware that not only simulates the major categories of real malware, but also allows adjusting its execution rate as well as its network-level activity.

We achieved high level of malware flexibility by allowing parameter specification in a configuration file. It determines malware type, its intensity in terms of execution progress, the network-level intensity in terms of data-packet sizes and interpacket delays. In all cases, our malware communicates with a C&C server to report its progress, upload data and confirm the completion of the individual malicious actions.

Logically our malware can be represented as a hierarchical structure. It starts execution by launching a dispatcher service that loads malware configuration and, based on the supplied parameters, launches other services, which serve as local dispatchers managing malware activities and synchronizing separate malware threads running concurrently.

Even if we fix some malware parameters in our experiments to avoid an explosion of the number of parameter settings, this is not a limitation of our synthetic malware; those parameters can be easily modified in a configuration file. The rest of this section describes individual services in details.

4.1 Info Stealers

Contacts. Our implementation of a contact stealer depends on two parameters: the number of contacts to extract and the delay. The process of collecting information consists of issuing multiple queries to different content providers. Delay specifies a time interval between consecutive queries, and in our experiments we varied it in the range 0 - 25 ms. 25 ms value adds approximately 1 sec per an access to a contact.

The other parameter, the number of contacts to be uploaded to a server, is meant to model the various amount of contacts stored on the phones. We chose the values of this parameter based on the concept of âĂIJcognitive limitâĂİ - the number of people with whom one can maintain stable social relationships. The anthropologists estimate âĂIJcognitive limitâĂİ to be in the range 150 - 290 [24], [28]. Statistical data show that an average mobile phone user has approximately 150 contacts [13], [15].

We generated contacts programmatically using the census data available online [14]. The length of people names was sampled from a normal distribution with parameters based on [14]. In our experiments, we varied the number of contacts to be stolen from 25 up to 250.

SMS. Analysis of real malware demonstrates that there exist two types of SMS stealers: batch stealers and active listeners (e.g. bank

[3]Click fraud MD5 5b5f65e1d014dbb8e55602ed468cb5eb

[4]MD5 b7f53ff767e130758a6375315e62e82a

trojans). The former upload all text messages to their C&C server, and the latter intercepts only some types of incoming text messages. We implemented both types of SMS stealers. As in the previous case, the phones storing different number of text messages are modeled by specifying how many messages to upload to the C&C server. We chose this parameter based on multiple studies of phone users' behavior [16], [17]. The college students send 2000 messages per month on average. Thus, we varied this parameter in the range 200 - 1700.

The other parameter, delay, slowdowns SMS stealing by suspending the SMS stealer in multiple points. In the experiments, we set the range of the delay parameter as 0 - 40 ms; the latter value adds 0.3 sec delay per SMS access.

The second flavor of SMS stealer, active listener, registers an Android listener, which must be supplied with the list of phone numbers and/or key words to search. If an SMS comes from a number in the list or its body contains one or more specified key words, it is intercepted. It is also possible to specify whether to upload intercepted messages to a server and/or remove them from the phone.

Location. There exist two ways of obtaining location: to get approximate location based on information available from cellular towers or get absolute coordinates using GPS module. We implemented both methods; if GPS is disabled, then approximate location is used. Our malware registers an Android listener to receive location updates. Obtaining location programmatically does not result in significant performance overhead because the last saved location is returned without accessing Android run-time environment. To slow down this, we can specify a corresponding delay.

File operations. Our malware can perform all basic file operations in response to commands received from C&C server. For example, the malware is able to download a file from a remote server and save it on an SD card. It can send directory content to the C&C server. After processing the directory listing, C&C server may request to upload a particular file. When uploading a file to the server, malware can read it at once or in chunks (the mode as well as chunk size are specified by C&C server). Between reading individual chunks of data or individual files and uploading them to the server, malware can pause itself for a predefined amount of time specified in the request.

Phone IDs and other info. We extract the broad set of available phone identifiers by accessing members of the static class android.os.Build: IMEI, several hardware ids, OS type etc. As in the previous case, the workload is negligible when measured through performance counters. In addition, our malware can extract the other types of sensitive data: information on installed apps, browser history and bookmarks.

4.2 Net node

Click fraud. Click fraud has become one of the most popular ways of generating revenue out of having free computational resources. Cybercriminals gather compromised hosts into botnets and make them click on the links located on the websites affiliated with attackers [20], [34]. The number of fraudulent clicks may reach 14% of the overall number of clicks on the Internet [6], [29].

The overhead incurred by click fraud mainly comes from fetching numerous webpages. Our click fraud implementation periodically accesses webpages specified in the supplied configuration file. To vary CPU workload across a wide range, we parallelized our algorithm, i.e. two threads fetches webpages concurrently.

In our experiments, we chose 20 URLs from the list of most popular websites [1] and varied the following two parameters: the total number of fetched webpages in the range from 20 up to 300, and the delay between the successive page accesses in the range 0 - 3 sec. The parameters were chosen based on the observation of the CPU activity when running our click fraud engine as a standalone process. The most intensive version of click fraud resulted in 30-36% CPU utilization, on the other extreme benign parameter settings consumed only 4-8% of CPU.

DDoS. The US-based DDoS protection company, Prolexic Technologies, has reported the growing number of mobile-based DDoS attacks [11]. It discovered that mobile devices participated in a large-scale multi-vector DDoS campaign in Q4 2013. The handsets were running either AnDOSid app or Low Orbit Canon (LOIC) app. Another anti-virus company, Dr. Web, detected in December, 2012 an Android malware [18] mounting DDoS attacks.

80% of DDoS attacks have been found to be performed using HTTP [26]. Thus, we implemented two HTTP level attacks: GET flood and slow-bandwidth attack (Slowloris [30]). The former attack sends a series of GET requests to a supplied list of webservers in the configuration file and waits for their responses. In comparison with GET flood, Slowloris requires much less computational resources and it is more favorable for mobile devices.

Low-bandwidth attacks try to exhaust server pool of available connections by opening numerous connections and sending data very slowly to keep connections alive. In our experiments, we opened 500 connections and specified 500 sec timeout. If some connections fail, we reopen them to keep the constant number of active connections. As a target host, we chose www.android.com. We conducted 4 experiments by varying the delay between opening consecutive connections from 1 ms up to 200 ms. To guarantee that the server does not close connections, our malware transmits small amount of data over established connections every second.

4.3 Compute node.

Computational workload comes in two flavors: brute-forcing SHA1 hashes and multiplying random matrices. Both activities let C&C server specify internal delay as well as parameters of the objects used in the computations. In the case of SHA1, C&C should specify the number of parallel threads, the maximum length of the original string, the maximum number of iterations and delay between consecutive iterations. In the case of matrix multiplication, the sizes of matrices and delay are defined by C&C sever.

4.4 Reflection and encryption

To hamper reverse-engineering, Android malware invokes Java methods through Java Reflection API and also encrypts all strings in the code. The use of reflection mechanism may lead to the appearance of extra strings in the code, this is why string encryption should accompany reflection. We implemented second versions of each malware that uses Java reflection and string encryption in the same way as the Geinimi.a sample [33].

5. EXPERIMENTAL SETUP

To study the impact of malware on the underlying apps, we selected 9 applications based on the diversity of their high-level behavior: computationally intensive, user-driven, and network-oriented. The chosen benign apps are: Firefox, Google Translate, Sana (medical app), TuneInRadio, Google Maps, Angry Birds, Zombie World War, CNN and Amazon. Morpheus currently includes 7 synthetic

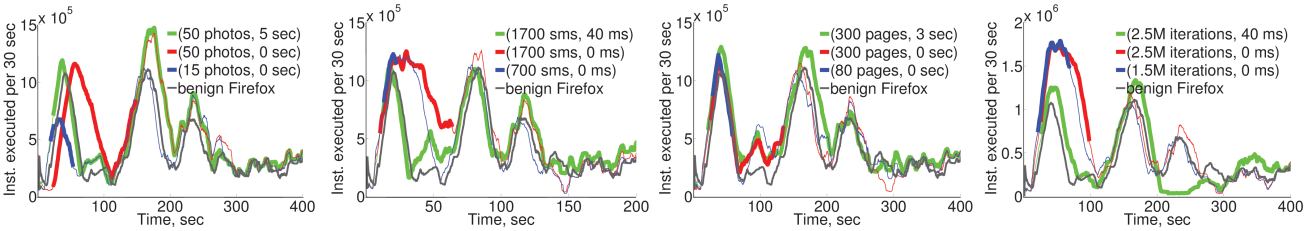

Figure 1: Firefox: instructions executed (different malware parameter settings).

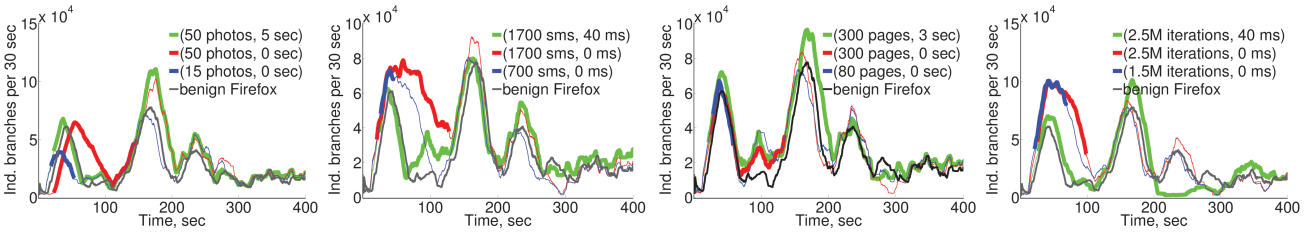

Figure 2: Firefox: indirect branches (different malware parameter settings).

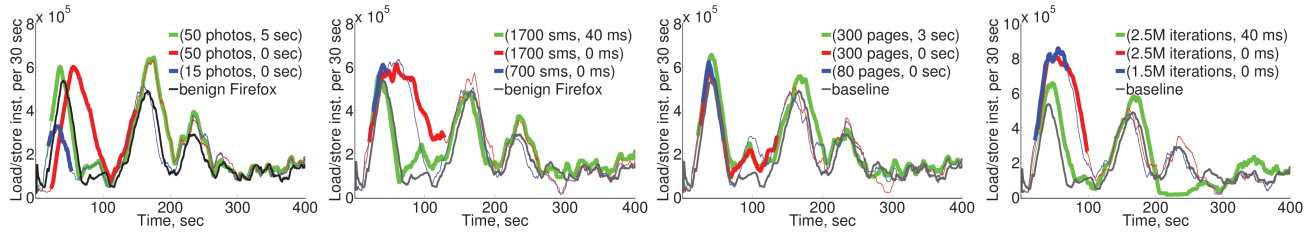

Figure 3: Firefox: load/store instructions (different malware parameter settings).

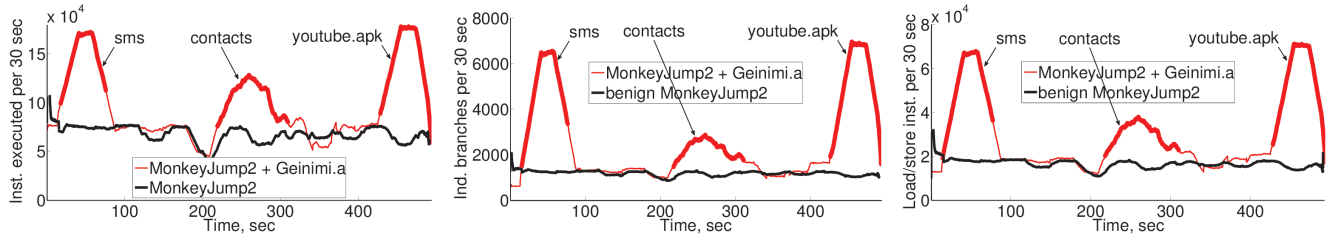

Figure 4: MonkeyJump2 vs MonkeyJump2 + Geinimi.a (real malware experiment).

malware families from the three high-level behaviors (66 malware samples including different parameter values) inserted into all 9 benign apps. Additionally, we repeated each experiment to study the effect of invoking methods through Java reflection and encrypting strings. We recorded 7.5 min long execution traces per configuration of synthetic malware and for each underlying benign app. Every execution trace is a collection of 7 performance counter time series.

In our experiments, we used two development boards: Samsung Exynos 5250 and TI OMAP 5430; each of them was running Android 4.2.1. The results are reported only for Exynos 5250 board. Automation of user events, when recording performance counter traces, was achieved by using the Android Reran tool [3] to replay the real user actions multiple times. Actual trace collection was done by ARM DS-5 v5.15 framework equipped with Streamline profiler, which is able to match OS-level events (e.g. context switches) with performance counter data.

We characterized app dynamic behavior by observing the following process-specific counters: the total number of executed instructions, the number of integer instructions, the number of direct and indirect branches, the number of load/store instructions and cycles. The only counter that was not process-specific that we used was the number of mispredicted branches.

We checked that synthetic malware executed correctly, including log messages produced by the Android middleware, incoming/outgoing network traffic generated by the board, markers sent to DS-5 tool, and responses sent to the C&C server (Hercules_3-2-6 TCP server in our experiments). In the case of real malware, we developed our own HTTP server supporting the custom protocol hardcoded in the malware. We instrumented both synthetic and real malware to make them report their actions both to adb console and to the DS-5 framework in order to synchronize high-level activities with performance counter traces.

6. EXPERIMENTAL RESULTS

To visualize malware impact on benign apps, we selected Firefox, which serves as an underlying benign app for repackaging purposes. Firefox is able to exhibit large variety of dynamic behaviors that covers all categories of our Android app classification. It behaves as computationally intensive app when running flash, it is similar to network intensive apps when streaming video, and eventually one can turn it into a user-driven app by reading news. We also chose 2 malware families (photo and SMS stealers) from information stealer category, one family (click fraud) from network node category, and the last malware family (md5 hash cracker) from the computational malware category.

Our experiments show that malware payload does affect performance counter traces, and that malware that uses reflection with string encryption makes hardware level analyses potentially *easier* even though it makes static analyses harder. At the same time, we demonstrate that baseline applications have very diverse behaviors and this can lead to false positives – proposed detection techniques should be designed carefully to minimize these.

6.1 Payload Parameters Affect Performance Counter Traces

Figures 1, 2, 3 shows the impact of 4 synthetic malware families on Firefox. The black line represents the underlying benign app's behavior. Within each malware family, we selected three configurations representing extreme values of the parameters controlling malware intensity and the size of malicious workload. The blue curve corresponds to the light malicious workload without artificial delays. The red line belongs to the malware that executed the largest workload in our experiments with no delays. In terms of malware intensity, i.e. the numbers of malicious operations executed per unit of time, both lines represent equally intensive malware samples. And finally the least intensive malware, carrying out the largest amount of work at the same time, is presented by the green curve.

All experiments involve the same sequence of user actions – we included browsing of the most popular websites (e.g. cnn.com, nytimes.com, google.finance etc) and streaming video on some websites (e.g. cnn.com). The curves shown on the plots are smoothed over a 30s window to highlight common trends.

In all experiments, malware starts its execution at a random moment within a time interval 15-45 sec. And it remains active from a few dozens of seconds up to several hundreds of seconds. The portion of a curve that corresponds to malicious activity is highlighted by thicker lines.

Analyzing the plots, we can easily notice that the impact of malware on performance counters depends not only on the intensity of malware, but also on the total amount of work malware carries out. However, intensity factor dominates. The initial part of the curves on a plot looks very much similar while malware remains dormant. As soon as it starts execution, the curve diverges from the benign one more and more.

Another interesting observation is that even after malware termination, there are still some residual effects, i.e. it takes time for the corresponding curve to resume following the original benign curve. Such residual effects likely come from the nature of the managed languages, i.e. malware affects the state of Dalvik virtual machine (VM). Relaxation time needs to pass before disturbance of Dalvik VM state disappears.

Further, it is not easy to predict the amount of disturbance a particular malware can cause. We plotted two similar malware types: photo and SMS stealer. In our experiments, the average photo size is about 4.5M, the average size of a text message is 51 symbols. If we compare impact of the most intensive SMS stealer with the impact of the most intensive photo stealer (red curves), we easily notice that stealing text messages is more noticeable process on the micro-architectural level. However, stealing 50 photos results in transmission of 225 Mb, and stealing 1700 sms messages leads to uploading 85 Kb data. The difference comes from the architecture of Android OS. To access just one text message requires issuing multiple queries to the corresponding content provider, while a photo as any other file can be accessed directly using the corresponding Java API.

If we estimate large-scale malware effects, we notice that malware mostly shifts the observed traces: it increases the amplitude, but only very intensive malware may change the way the traces look like. Some variations of the presented curves should be attributed to the constantly changing layout of the webpages. Thus, the same sequence of user actions might cause rendering slightly different webpages. However, such variations do not change the overall picture.

Finally, we illustrate Geinimi.a's impact (Fig. 4) on the underlying MonkeyJump2 game. Some malicious activities (e.g. app installation) drammatically changes the performance counter traces, while the rest remain unnoticeable (e.g. stealing phone IDs).

6.2 Java Reflection Potentially Eases Detection

When experimenting with synthetic malware augmented with reflection and string encryption, we discovered that reflection and accompanying it encryption make malware much more visible at the micro-architectural level (Fig. 5). We present results for AngryBirds, not for Firefox because Android Reran had trouble adapting to webpage layout changes due to appearance of ad in random spots.

We see that even the least intensive malware in our collection significantly increases the intensity of micro-architectural events and makes the gap between benign and malicious traces wider. As we mentioned before, traces coincide (in this case, they are perfectly aligned), while malware has not started its execution. The large spike on the two traces results from displaying flash-based ad, thus they should be excluded from consideration.

Reflection leads to performance overhead because of the extensive use of dynamically resolved types in the code. This prevents VM from performing some optimizations, and thus increasing CPU workload [7]. Reflection and string encryption lead us to conclude that analysis of CPU activity may reveal information that is hidden from static code analysis tools.

6.3 Diversity of Benign Traces May Cause False Positives

Besides analyzing malware impact on benign traces, we compare micro-architectural fingerprints of benign applications belonging to different categories of our classification. To illustrate what we discovered we plot traces corresponding to the different runs of the following benign apps: Firefox and Zombie World War (Figures 6, 7). Surprisingly, the latter app demonstrates the steadiest behavior. To double check this observation, we compared traces of other computationally intensive apps and we saw the same trend. This can be explained as follows: user interactions do not significantly change the amount of work the app executes in the background

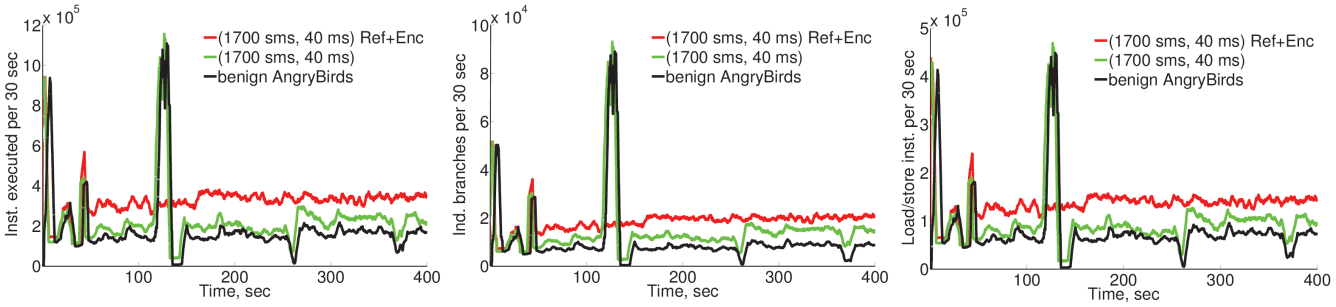

Figure 5: AngryBirds: malware vs malware + reflection & encryption (effects of malware obfuscation).

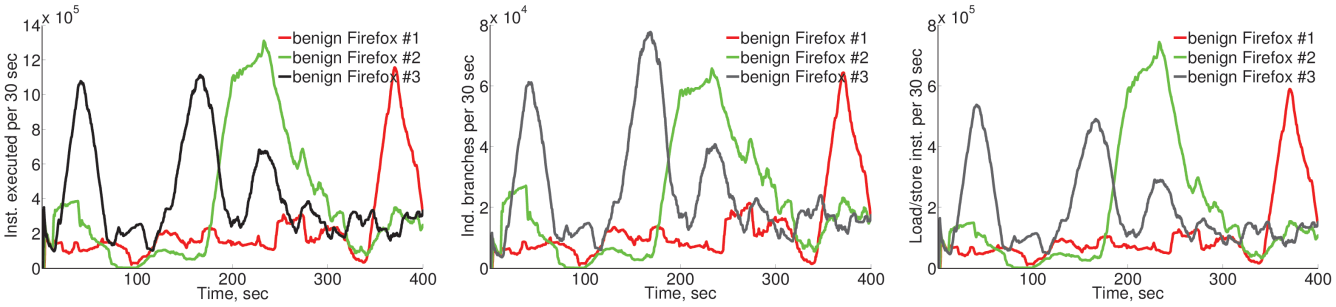

Figure 6: Firefox: diversity of benign traces (potential source of false positives).

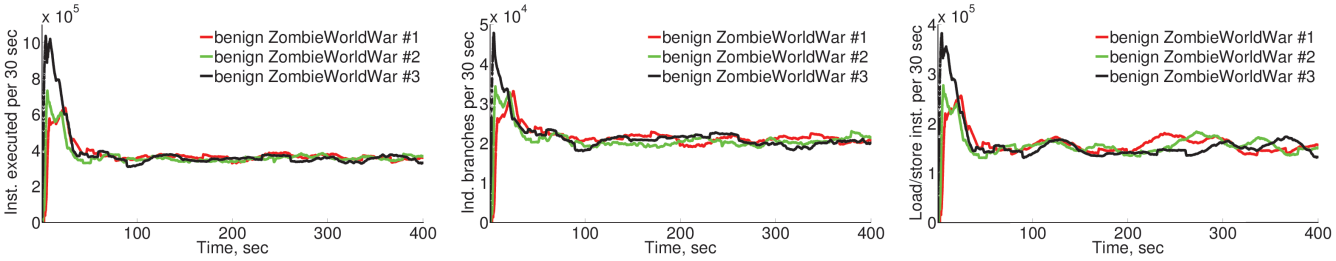

Figure 7: ZombieWorldWar benign: diversity of benign traces (potential source of false positives).

when doing rendering and carrying out other game-related activities. Micro-architectural traces of the browser app, Firefox, are more event-sensitive: we see large spikes resulting from the input events.

This experiment highlights the fact that micro-architectural behavior of applications may change in a broad range depending on user input. Such poorly predictable changes could lead to the high level of false positives in the systems that try to detect malware observing performance counters.

7. RELATED WORK

Performance counters. Performance counters not only allow non-intrusive monitoring of applications performance, but also may help to deduce some high-level information. Ammons et al. [19] demonstrate an approach to flow sensitive profiling, which establishes correspondence between hardware performance metric and individual execution paths. Azimi et al. [21] build a system for run-time analysis of applications' performance using performance counters. Bare et al. [22] investigate possibilities for performance analysis in distributed systems. SimPoint project [31], [32] introduces a notion of execution phases and demonstrates different ways to identify them: one of the possibilities is to use PMU.

Native code. CFIMon [35] enforces control-flow integrity using performance counters along with binary code analysis. Malone et al. [27] show how to accomplish static and dynamic code integrity checking using PMU. They fit linear models to PMU data and attach model description to the corresponding files. Yuan et al. [36] apply PMU to the detection of common control-flow hijacking attacks.

Android. In comparison with desktop operating systems, Android adds extra level of complexity by running apps within a Dalvik virtual machine (VM). Zhou et al. [37] comprehensively analyzed Android malware, and they found that most malware samples (86% out of 1,200 samples) in their dataset were repackaged versions of benign apps. Demme et al. [23] postulate the possibility of using performance counters for detection of known Android malware families.

8. CONCLUSIONS

Most Android malware samples are repackaged version of the benign apps. Thus, when evaluating detection schemes, one needs to estimate their true and false positive rates by testing them on the underlying benign apps, and not only on random benign apps. Also, we propose to go one step further and include malware pay-

load adaptations to test such detection methods. In this paper, we propose a tool to evaluate the robustness of hardware based approaches for malware detection and we show that easy changes to malware intensity or implementation, while maintaining the same semantics, may significantly affect micro-architectural traces.

9. ACKNOWLEDGMENTS

The authors would like to thank Chris Grier (ICSI) and Paul Pearce (UC Berkeley) for discussions on malware and classifying their behavior, and the anonymous reviewers for their considerate feedback. This work was partially supported by National Science Foundation Grant No. CNS-1314709.

10. REFERENCES

[1] 500 top webites. http://www.alexa.com/topsites.

[2] Android malware blog. http://contagiodump.blogspot.com.

[3] Androidreran. http://www.androidreran.com.

[4] Exploid exploit. http://forum.xda-developers.com/showthread.php?t=739874.

[5] Gingerbreak exploit. http://droidmodderx.com /gingerbreak-apk-root-your-gingerbread-device.

[6] How fictitious clicks occur in third-party click fraud audit reports. http://static.googleusercontent.com/media/www.google.com/en/us /adwords/ReportonThird-PartyClickFraudAuditing.pdf.

[7] Java reflection. http://docs.oracle.com/javase/tutorial/reflect.

[8] Malware database. http://virusshare.com.

[9] Mobile bitcoin miner. https://blog.lookout.com/blog/2014/04/24/badlepricon-bitcoin.

[10] Mobile malware evolution: 2013. https://www.securelist.com /en/analysis/204792326/Mobile_Malware_Evolution_2013.

[11] Prolexic q4 2013 report. http://www.prolexic.com/knowledge-center-ddos-attack-report-2013-q4.html.

[12] Universal android rooting procedure. http://theunlockr.com /2010/10/26/universal-android-rooting-procedure-rage-method.

[13] Average number of phone contacts (us), 2010. http://www.mediapost.com/publications/article/122938.

[14] Us census, 2010. http://www.census.gov/2010census.

[15] Average number of phone contacts (uk), 2011. http://www.10yetis.co.uk/releases/average-brit-has-476-facebook-friends-compared-to-152-mobile-phone-contacts 402.html.

[16] Average number of sms, 2011. http://www.phonedog.com/2011/12 /16/how-many-text-messages-do-you-send-per-month-on-average.

[17] Average number of sms, 2011. http://www.pewinternet.org/2011/09/19/how-americans-use-text-messaging.

[18] Android trojan mounting ddos attack, 2012. http://news.drweb.com /show/?i=3191.

[19] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *ACM Sigplan Notices*, 1997.

[20] K. Asdemir, O. Yurtseven, and M. Yahya. An economic model of click fraud in publisher networks. *Journal of Electronic Commerce*, 2008.

[21] R. Azimi, M. Stumm, and R. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *International Conference on Supercomputing*, 2005.

[22] K. Bare, S. Kavulya, and P. Narasimhan. Hardware performance counter-based problem diagnosis for e-commerce systems. In *International Conference on Supercomputing*, 2010.

[23] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, A. Aethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. In *International Symposium on Computer Architecture*, 2013.

[24] R. Dunbar. Neocortex size as a constraint on group size in primates. *Journal of Human Evolution*, 1992.

[25] P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Computer and Commnications Security*, 2011.

[26] D. Holmes. The ddos threat spectrum, 2012. http://www.f5.com/pdf/white-papers/ddos-threat-spectrum-wp.pdf.

[27] C. Malone, M. Zahran, and R. Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *ACM Workshop on Scalable Trusted Computing*, 2011.

[28] C. McCarty, P. Killworth, H. Bernard, E. Johnsen, and G. Shelley. Comparing two methods for estimating network size. In *Applied Anthropology*, 2001.

[29] A. Metwally, D. Agrawal, and A. Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. In *International conference on World Wide Web*, 2007.

[30] RSnake and J. Kinsella. Slowloris http dos, 2009. http://ckers.org /slowloris.

[31] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 2003.

[32] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *International Symposium on Computer Architecture*, 2003.

[33] T. Strazzere and T. Wyatt. Geinimi trojan technical teardown. 2011. https://blog.lookout.com /_media/Geinimi_Trojan_Teardown.pdf.

[34] A. Tuzhilin. The lane's gifts v. google report. 2006. http://googleblog. blogspot.com/pdf/Tuzhilin_Report.pdf.

[35] Y. Xia, Y. Liu, H. Chen, and B. Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *Dependable Systems and Networks*, 2012.

[36] L. Yuan, W. Xing, H. Chen, and B. Zang. Security breaches as pmu deviation: Detecting and identifying security attacks using performance counters. In *Asia-Pacific Workshop on Systems*, 2011.

[37] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy*, 2012.