Quantifying and Improving the Efficiency of Hardware-based Mobile Malware Detectors

Mikhail Kazdagli
UT Austin
mikhail.kazdagli@utexas.edu

Vijay Janapa Reddi
UT Austin
vj@ece.utexas.edu

Mohit Tiwari
UT Austin
tiwari@austin.utexas.edu

***Abstract*—**

Hardware-based malware detectors (HMDs) are a key emerging technology to build trustworthy systems, especially mobile platforms. Quantifying the efficacy of HMDs against malicious adversaries is thus an important problem. The challenge lies in that real-world malware adapts to defenses, evades being run in experimental settings, and hides behind benign applications. Thus, realizing the potential of HMDs as a small and battery-efficient line of defense requires a rigorous foundation for evaluating HMDs.

We introduce Sherlock—a *white-box* methodology that quantifies an HMD's ability to detect malware and identify the reason why. Sherlock first deconstructs malware into atomic, orthogonal actions to synthesize a diverse malware suite. Sherlock then drives both malware and benign programs with real user-inputs, and compares their executions to determine an HMD's *operating range*, i.e., the smallest malware actions an HMD can detect.

We show three case studies using Sherlock to not only quantify HMDs' operating ranges but design better detectors. First, using information about concrete malware actions, we build a discrete-wavelet transform based *unsupervised* HMD that outperforms prior work based on power transforms by 24.7% (AUC metric). Second, training a *supervised* HMD using Sherlock's diverse malware dataset yields 12.5% better HMDs than past approaches that train on ad-hoc subsets of malware. Finally, Sherlock shows *why* a malware instance is detectable. This yields a surprising new result—obfuscation techniques used by malware to evade static analyses makes them more detectable using HMDs.

I. Introduction

Mobile devices store personal, financial, and medical data and enable malicious programs to spread quickly through app-stores. Unsurprisingly, 2015 saw 900K new mobile 'malware' compared to 300K in 2014. Mobile malware infects applications through errors by users, developers, or platforms like Android [1], [2], [3]. Once infected, malware can run 'payloads' such as stealing private data from the victim device or making HTTP requests to attack a remote server while masquerading as the infected application. Hence, machine learning classifiers that differentiate operating system and network *behaviors* of benign programs from malware are an attractive line of defense against mobile malware [4].

Hardware-based malware detectors (HMDs) are a recent category of behavioral malware detectors [5], [6], [7], [8]. An HMD observes programs' instruction and micro-architectural traces and raises an alert when the current trace's statistics look either anomalous compared to benign traces (unsupervised HMDs) or similar to known malicious traces (supervised HMDs). HMDs are small, secure even from a compromised OS, and can observe instruction-level attacks (such as row hammer [9], [10] and side-channel data leaks [11]) that leave no system call trace [12]. HMDs are thus a trustworthy first-level detector in a network-wide malware detection system [13], [14] and are being deployed in commercial mobile devices[1] as of early 2016.

978-1-5090-3508-3/16/$31.00 ©2016 IEEE

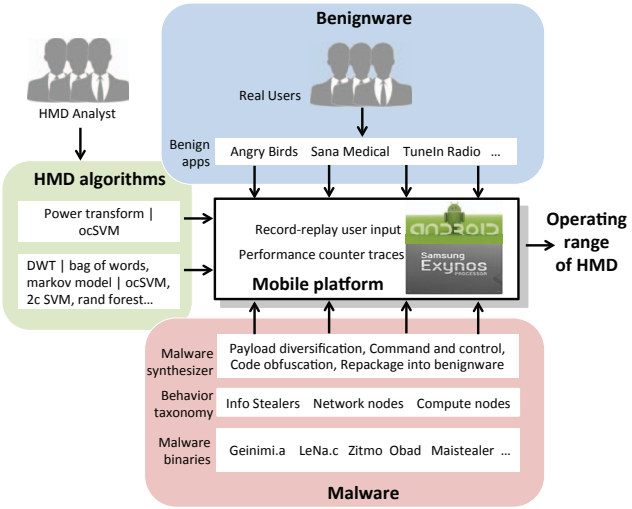

Fig. 1. Overview of Sherlock.

Architects designing HMD accelerators face two unique challenges. First, unlike benign programs like SPEC, malware *adapts* to proposed modeling algorithms and will evade detectors that only learn behaviors of existing malware [15]. For example, we found that changing only the number of execution threads or inter-action delays in a malware was sufficient to evade an HMD [5] trained on single-threaded binaries of the same malware. Second, to compare malware against benignware executions, HMDs have to run both programs with *real user inputs*. For example, a human user playing Angry Birds produces very different instruction traces compared to quiescent traces when no user is driving the app. Hence, the traditional 'black-box' approach for evaluating HMDs – without looking deeply into malware computation and without real user interaction – yields results that will not hold in practice.

In this paper, we present Sherlock—a 'white-box' meth–odology to evaluate HMDs for mobile malware. Sherlock is built on two principles: (1) malware will adapt to evade detection,

[1]https://www.qualcomm.com/products/snapdragon/security/smart-protect

and (2) malware hides behind benign programs, and only by running both malware and benignware with real user-inputs can we determine whether an HMD can tell them apart. These principles lead to a significant system-building effort and to new insights about HMDs for mobile malware.

The Sherlock platform in Figure 1 embodies both principles: (1) Sherlock synthesizes malware specifically to find the breaking point of an HMD under test. To do so, we introduce a taxonomy of mobile malware and present a synthesis tool that generates obfuscated malware with a configurable payload (i.e., tasks to run) that is a superset of the 229 malware we studied. (2) Sherlock tests HMDs when benign and malware programs use the same, long-running user inputs. To do so, Sherlock correctly records and replays thousands of 5–10 minute long user sessions (such as playing Angry Birds or running medical diagnostics) on real hardware. An HMD analyst can then use Sherlock's third component – HMD algorithms – to design and evaluate new ways of extracting features from program traces in order to train machine learning algorithms.

Sherlock's design principles yield a new metric for quantifying HMDs' performance. An *operating range* of an HMD algorithm is a metric that tells an analyst the root cause behind a malware alert as well as when the HMD fails. An operating range is expressed as *the smallest malware payload X hidden in application Y that an HMD algorithm A can detect with a false positive rate of Z*. For example, an analyst can determine how efficiently a compromised browser (Y) can steal SMSs or photos (X) when a random-forest HMD (A) is deployed at a pre-set false positive rate of 5% (Z).

The operating range of an HMD is independent of the training and testing set of malware – instead, it is defined in terms of atomic actions in malware payloads (X) such as stealing one photo or an SMS, sending an HTTP request, etc. An analyst can thus use the operating range to quantify HMD performance based only on (relatively invariant) high-level malware behaviors. Further, operating range describes false positive rate Z by comparing malware to the exact benign app Y that malware hides in—comparing a malware run to an arbitrary benign app or system utility yields an unrealistically good false positive rate.

Case Studies using Sherlock. We demonstrate Sherlock's utility by designing better HMDs than prior work, and by showing (for the first time) that evading static program analysis makes malware more visible to HMDs.

Our first case study shows that taking concrete mobile malware actions into account yields a better unsupervised HMD than directly applying desktop HMDs designed to detect short-lived exploits. Specifically, atomic software-level actions on mobile devices such as stealing a 4MB photo or one SMS takes a long time at the hardware level (2.86s and 0.12s respectively on a Samsung Exynos 5250 device). We design a new HMD that uses longer-duration (100ms) feature vectors, extracts low-frequency signals, and is 24.7% more effective using the area under the ROC curve (AUC) metric than prior work [6].

Our second case study uses Sherlock's malware synthesizer to design supervised HMDs with 97.5% AUC – 12.5% better than prior work. Specifically, we train on a malware set that covers diverse, orthogonal behaviors compared to prior work that trains HMDs on an ad-hoc subset of behaviors. Further, the supervised HMD's operating range covers even small data (1 photo, 25 contacts, 200 SMSs, etc) being stolen with close to 100% accuracy at a 5% false positive rate. However, malware payloads that clog remote servers by sending them HTTP requests are virtually undetectable at the hardware level—Sherlock provides such semantic insights into why HMDs succeed and fail.

Our final case study in using Sherlock's malware synthesizer yields a surprising result—*obfuscation techniques that evade detection by static analysis tools make HMDs more effective.* Specifically, malware developers use string encryption and Java reflection to create high-fanout nodes in data- and control-flow graphs and thus foil static analysis tools. However, these obfuscation techniques in turn create instruction sequences and indirect jumps that make malware stand out from benignware. Hence, light-weight HMDs can complement static analysis tools [16] used by Google and other app stores to drive malware down into more inefficient settings.

In summary, we make the following contributions:

1. Malware Synthesis. We deconstruct 229 malware binaries from 2013–2015 to create a malware synthesis tool. An analyst can use the synthesis tool to determine an HMD's operating range.

2. Record-and-replay Platform. Sherlock records and replays 1–2 hours each of real human input for 9 benign applications and over 69 hours across 594 malware binaries. Without correct replay at these time-scales, malware payloads will not execute to completion.

4. Three case studies with new insights. We improve HMDs' performance by 24.7% and 12.5% respectively for unsupervised and supervised HMDs and show that HMDs detect stealthy malware that evades static analysis tools.

Sherlock Detailed information on how to reproduce the experiments can be found at https://github.com/Sherlock-2016.

II. Motivation

Before we dive into the details of Sherlock in Sections III and IV, we begin with the unique advantages of HMDs over OS-level detectors and challenges in evaluating HMDs.

A. HMDs in a Network of Weak Detectors

HMDs (as well as OS-level detectors) are deployed in a collaborative intrusion detection system (CIDS) that has two components. On the server side, a platform provider (e.g., Google) executes benign and/or malware applications using test and real user inputs, measures instruction statistics (using performance counters for example), and creates a database of computational models. On client devices, a light-weight *local detector* samples performance counters to create run-time traces from applications, compares each run-time trace to database entries on the device, and forwards suspicious traces to a *global detector* on the server.

Importantly, HMDs do *not* need to have 0% false positives and 100% true positives—they only need to serve as an effective

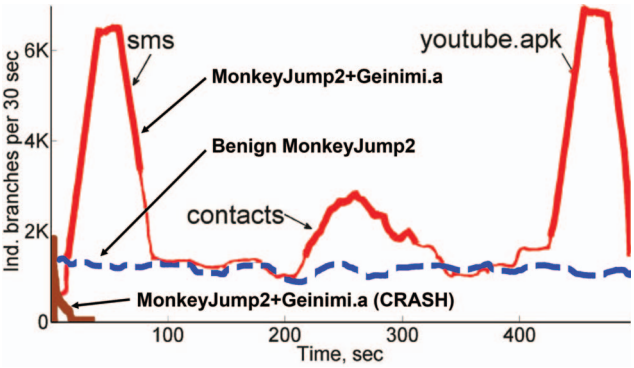

Fig. 2. Executing malware payloads. The off-the-shelf Geinimi.a malware crashes immediately. Once fixed, Geinimi.a executes malicious payloads such as stealing SMSs or contacts or downloading files.

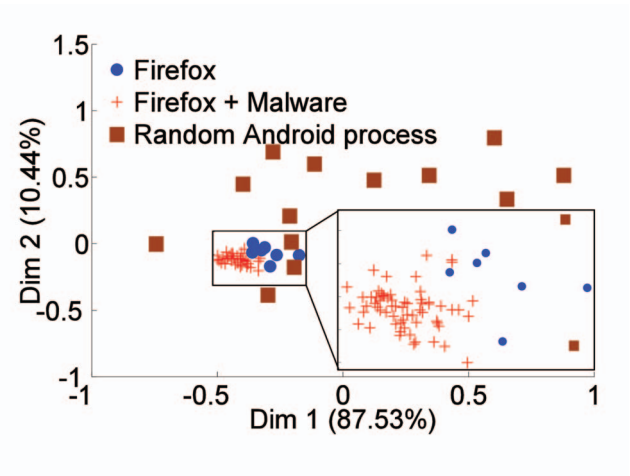

Fig. 3. Differential analysis of malware v. benignware. The plot shows principal components of benign Firefox, Firefox with malware, and arbitrary Android apps. Malicious Firefox's traces are closer to Firefox than to random apps.

filter for a global detector that can then use program analysis [17], [18] or network-based algorithms [19], [20] to build a robust global detector. We refer readers to Vasilomanolakis et al. [13] for a survey on collaborative malware detectors.

B. Hardware vs OS-level detectors

HMDs are more trustworthy, light-weight, and hard to hide from compared to detectors that use system call [4], [21], middleware [22], or network based behavioral analysis [23],

HMDs' are trustworthy since they can be isolated from most of the OS (and Android middleware) and run inside a hardware-based enclave [24], [25] or directly in hardware [7] – secure against even user errors and kernel rootkits [3]. HMDs can be battery efficient with feature extraction and detection logic implemented using accelerators [7]. Finally, HMDs can detect malware that leaves *no* system call trace – such as rowhammer [9], [10], A2 [26], and side-channel attacks [12] on desktops and, as we show in this paper, evasive mobile malware that hides behind benign applications and requests no additional sensitive permissions.

Interestingly, on mobile platforms, HMDs have comparable detection rates to OS-level detectors (although we leave details of this comparative experiment out of this paper). We find that OS-level detectors that model system calls also reach detection rates of almost 90% at false positive rates close to 10%. This is close to our HMDs' performance (in Section 5)—as a result, HMDs can not just be more trustworthy than OS-level detectors but be competitive in detection performance as well.

HMDs for desktops do not directly port over to mobile platforms. Ozsoy et. al's [7] hardware-accelerated classifiers detect ∼90% of off-the-shelf desktop malware with 6% false positive rate. Tang et. al's [6] anomaly detector achieves 99% detection accuracy for less than 1% false positives on a set of PDF and Java malware. Such desktop HMDs, however, do not work well for mobile platforms – we quantify Tang et. al's HMD against mobile malware in our first case study in Section 5 and build a 24.7% better HMD using Sherlock.

C. Challenges in Evaluating HMDs

The closest related work to ours – on HMDs for mobile malware – is by Demme et al. [5], where the authors present a supervised learning HMD that compares off-the-shelf Android malware to arbitrary benign apps, yielding a 90:10 true positive to false positive ratio. However, this methodology of using off-the-shelf malware and comparing it to arbitrary benign apps is fallacious, as we discuss next.

Adaptive malware. One challenge in evaluating detectors is that malware developers can *adapt* their apps in response to proposed defenses. For example, we find that simply splitting a payload into multiple software threads dramatically changes the malware's performance-counter signature and training a supervised HMD on the single-threaded execution yields a very low probability of labeling the multi-threaded version as malware. Adding delays, changing payload intensity, or choosing an alternative victim application also throws off a supervised HMD trained only on traces from existing malware.

Prior work analyzes malware samples categorized by family names like `CruseWin` and `AngryBirds-LeNa.C`—this does not tell an analyst why a malware binary was (not) detected. Instead, we propose to determine *why* a particular malware sample was (un)detectable, to anticipate *how* it can adapt, and then to create a malware benchmark suite to *identify the operating range* of the detector.

Correct execution. The second challenge is that mobile malware samples available online [27], [28], and used in prior work, seldom execute 'correctly'. Malware often require older, vulnerable versions of the mobile platform, they may target specific geographical areas, include code to detect being executed inside an emulator, wait for a (by now, dead) command-and-control server to issue commands over the internet or through SMSs, or in many cases, trigger malicious actions only in response to specific user actions [29], [30].

Figure 2 shows that an off-the-shelf malware (Geinimi.a) simply crashes on our Android board and thus looks distinct from a benign MonkeyJump game's trace. Prior work will misclassify this as a true positive. 20% of malware executions in Demme et al's [5] experiments lasted less than one second and 56% less than 10 seconds – in comparison, stealing a single photo takes almost 3 seconds. Instead, we ensure that malware executes 'correctly' – steals SMSs and contacts, and downloads

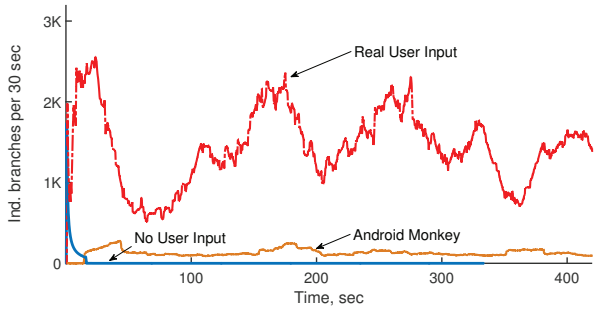

Fig. 4. Real user inputs create hardware level activity, while providing no input or using Android's input-generation tool (Monkey) creates a very small signal.

an app – and aim to identify these payloads.

Appropriate Benignware and Real User Inputs. The third challenge is to ensure appropriate differential analysis between benign and malware executions. Prior work [5] trains detectors on malware executions but tests against *arbitrary* benign applications. However, a benign app infected with malware looks more similar to the underlying benign app than an arbitrary benign app. Figure 3 plots the execution traces of Firefox, Firefox with malware, and randomly chosen Android processes along the first two principal components that retain $\sim 99\%$ of the signal. We see that the infected Firefox traces are much closer to those of benign Firefox than to any other Android process like `netd`. Hence, false positive rate of an HMD for Firefox should be tested using a benign Firefox – testing the HMD against arbitrary processes [5] will yield wrong results that favor the HMD.

Further, Figure 4 shows that driving Android applications using real user-input (red curve) has a major impact on the execution signals compared to giving no input (blue curve) or using the Android 'Monkey' app (light brown curve) to generate random inputs. Behaviors with 'no inputs' or 'Android Monkey' (blue and brown curves) can be easily captured by a behavioral detector, and, as in the previous case, this leads to overestimation of its actual detection performance. Hence, we propose to test HMDs using malicious binaries against appropriate benign apps while both apps are being driven using real user-inputs.

Quantitative Comparison to Prior Evaluation Methods. We have shown in this section that prior 'black-box' methods yield traces that do *not* represent either malware or benignware executions. The prior method has logical flaws – as a result, 20% of malware traces in [5] are shorter than 1 second, and 56% are <10s – and we deliberately eschew further quantitative comparisons with Sherlock. Instead, our evaluation focuses on case studies using Sherlock to yield new insights into building effective HMDs.

III. Synthesizing Mobile Malware

The first major component of Sherlock generates a diverse population of malicious apps. To do so, we first introduce a taxonomy of high-level malware behaviors, and then use it to create a set of representative malware whose hardware signals have been explicitly diversified.

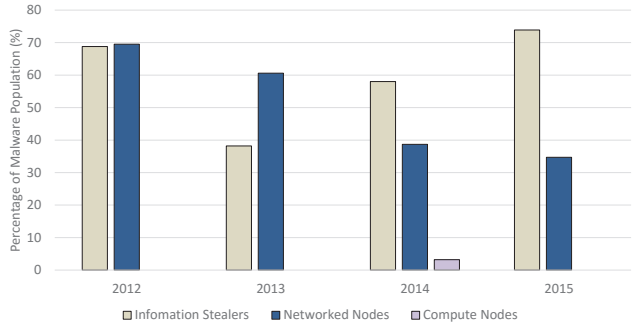

Fig. 5. Malware behaviors observed in a 126-family 229-sample Android malware set from Contagio minidump. Most malware steals data or carries out network fraud. However, samples that use phones as compute nodes, e.g., to crack passwords or mine bitcoins, have been reported in 2014.

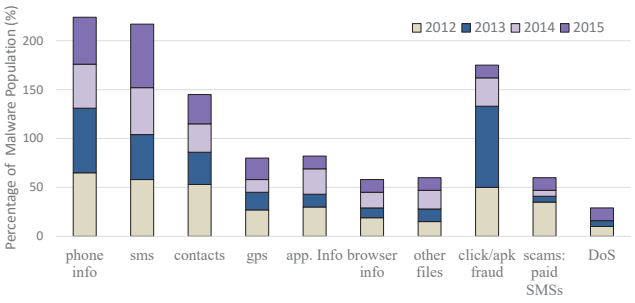

Fig. 6. Examples of malware behaviors and their contribution to the malware dataset.

Figures 5 and 6 show our manual classification of malware into high level behaviors. We studied 53 malware families from 2012, 19 from 2013, 31 from 2014 and 23 from 2015 – a total of 229 malware samples in 126 families – downloaded from public malware repositories [28], [31], [32]. Our classification's goal is to identify orthogonal atomic actions and to determine concrete values for these actions (e.g., amount and rate of data stolen).

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

A. Unique Aspects of Mobile Malware

Our key insight is that instead of trying to detect conventional root *exploits* [33], [34], [35], we propose to detect malicious *payloads*. Here, payloads refer to code that achieves the malware developers' goals, such as sending premium SMSs, stealing device IDs or SMSs, etc. We observed root exploits in only 10 of 143 samples in 2012 and 3 of 32 samples in 2013 – we now take a closer look at the attack vectors mobile malware rely on.

Mobile malware can successfully execute payloads due to vulnerable third-party libraries. In one instance that affected hundreds of millions of users, a "vuln-aggressive" ad-library had a deliberate flaw that led to downloaded files being executed as code [2]. Webviews, that enable Android apps to include HTML/javascript components, are another major source of

vulnerabilities [36] that allows payloads to be dropped to a device. Apps with this vuln-aggressive library or Webviews are otherwise benign and can be downloaded from app stores as developer signed binaries, only to be compromised when in use.

In other cases, errors by an app's benign developers themselves can lead to malicious payloads being executed. Misconfigured databases even in popular apps like Evernote [37] and AppLocker [38] (a secure data storage app) were vulnerable to malicious apps on the device simply reading out data from sensitive databases. In such cases, the malicious app could be an otherwise harmless wallpaper app that constructs an 'Intent' (a message) to AppLocker's database at run-time and exfiltrates data if successful.

User errors are another cause for malware payloads executing successfully at run-time. Malicious apps read data from an online server, use it to construct a user prompt at run-time, and thus request sensitive permissions such as access to SMSs or microphone. Users often accept such requests [39] and once authorized, apps can siphon off *all* SMSs or conduct persistent surveillance attacks [40].

Worst of all, even the platform (Android) code can have severe vulnerabilities that doesn't require a conventional exploit. For example, the Master Key vulnerability [3] simply involved an error in how Android resolves a hash collision due to resource-names in a binary at install time v. execution time. By packing the binary with a malicious payload such that the install time check passes but the execution time loader picks the other malicious payload, attackers could distribute their payloads through signed apps in official app-stores.

Finding: analyze payloads instead of exploits. We conclude that while there are many routes to getting a payload to execute as part of a benign app, executing the payload is mandatory for malware to win. Hence mobile HMDs shoud aim to *distinguish malicious payloads from benign app executions*. The challenge of detecting payloads is that payloads can look very similar to benign app's functionality. For example, if a previously harmless AngryBirds game starts to comb through a database, can we distinguish whether it is reading a user's gaming history (harmless) or a user's SMS database (attack) using only hardware signals.

B. Behavioral Taxonomy of Mobile Malware

At a high level we assigned every malicious payload to one or more of three behaviors: *information stealers, networked nodes*, and *compute nodes* (Figure 6).

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.

Synthetic Malware	Parameters (number of items)	Malware-Specific Delay (ms)	# of RPKG Mal. Apks	Length per Action (sec)	Inst. Count (Million)
Steal files (4.2MB each)	1, 15, 35, 50	0, 1K, 5K	12	2.86	50.97
Steal contacts	25, 70, 150, 250	0, 10, 25	12	0.36	67.80
Steal SMSs	200, 400, 700, 1.7K	0, 15, 40	12	0.12	25.90
Steal IDs, GPS	data size fixed	0, 200	2	4*	39.65
Click fraud (pages)	20, 80, 150, 300	0, 1K, 3K	12	0.40	44.40
DDos (slow loris)	500 connections	1, 40, 80, 200	4	425	49.70
SHA1 pass. cracker	10K, 0.5M, 1.5M, 2.5M	0, 20, 40	12	2.8E-5	1.9E-2

Fig. 7. Malware payloads: 4 info stealers, 2 networked nodes, and 1 compute node. These settings represent a small but computationally diverse subset of malware behaviors. Interestingly, small software actions have large hardware footprints.

Given the advances in mobile processors, we anticipated a new category of malware that would use mobile devices as compute nodes; for instance, mobile counterparts of desktop malware that runs password crackers or bitcoin miners on compromised machines. This was confirmed by recent malware that mines cryptocurrencies [41]. We use a password cracker as a compute-oriented malware payload. The cracker's task is to recover sensitive passwords by making a guess, compute the guess' cryptographic hash, and compare each hash against a secret database of hashed passwords.

Finding: Software-level actions are surprisingly long in hardware. Figure 7 shows the specifics of each malware behavior we currently include in Sherlock. Interestingly, *atomic* malware payload actions take significant amount of time at the hardware level for several payloads – e.g., stealing even one SMS or a Contact requires 0.12s to 0.36s on average. These constants inform the design of our performance counter sampling durations and machine learning models in Section IV. The last two columns in Figure 7 show the average length of an atomic action in the malware payload (not counting delays such as being scheduled out by the operating system), and the instruction count per action (e.g. stealing 1 photo/contact/SMS, clicking on 1 webpage in click fraud, opening 500 connections and keeping them alive in a DDoS attack, generating 1 string and computing its hash using SHA1).

C. Constructing Malware Binaries

We now describe the steps required to create a realistic malware binary. Malware activation can be chosen from being triggered at boot-time, when the repackaged app starts, as a response to user activity, or based on commands sent over TCP by a remote command and control (C&C) server. In all cases, malware communicates back to the C&C server to transfer stolen data or compute results. Sherlock's configuration parameters also specify network-level intensity of malware payload in terms of data packet sizes and interpacket delays, and device-level intensity in terms of execution progress (in terms of malware-specific atomic functions completed). We

App name	Description	# of Installs	User Actions	User Time (min)	CPU Time (min)	Inst. Count (Billion)
Amazon	internet store	10M – 50M	searched for sporting goods; looked through 25 pages; clicked on 50 items	81.15	32.40	1,914.97
Angry Birds	game	1M – 5M	played 9 rounds and completed 7 levels	76.97	63.76	1,047.73
CNN	news app	5M – 10M	browsed several categories of news and a few articles of each type	58.04	11.60	254.85
Firefox	browser	50M – 100M	browsed 20 webpages starting from google.finance	93.96	45.51	1,464.52
Google Maps	map service	500M – 1B	browsed maps of a few cities and opened street views	56.09	35.38	768.31
Google Translate	translator	500M – 1B	translated 30 words, searched history, tried handwriting recognition	59.72	12.12	203.61
Sana MIT Medical	medical app	U/A	completed 5-6 questionnaires	111.41	11.37	145.94
TuneIn Radio	internet radio	50M – 100M	switched amongst 6 channels and listened to radio	78.10	26.17	407.99
Zombie WorldWar	game	1M – 5M	played 5 rounds and completed 4 levels	91.62	88.40	2,261.99

Fig. 8. Real user inputs on benign apps, with per app traces up to ∼2 hours and ∼2 trillion instructions. We choose complex apps and include a mix of compute (games), user-driven (browsers, medical app), and network-centric (radio) apps.

chose concrete parameters for malicious payload based on an empirical study of mobile malware as well as information about benign mobile devices [42].

The generated malware has a top-level dispatcher service that serves as an entry point to the malicious program; it parses the supplied configuration file, launches the remaining services at random times, and configures them. Malicious services can run simultaneously or sequentially depending on the configuration parameter. In some cases, the service that executes a particular malicious activity can serve as an additional dispatcher. For example, the service executing click fraud spawns a few Java threads to avoid blocking on network accesses. Every spawned thread is provided with a list of URLs that it must access. Besides Android services, we register a listener to intercept sensitive incoming SMS messages, forward them to C&C server, and remove them from the phone if needed. This listener simulates bank Trojans that remove confirmation or two-factor authentication messages sent by a bank to a customer.

Most professional apps are obfuscated using Proguard [43] to deter plagiarism. Proguard shrinks and optimizes binaries, and additionally obfuscates them by renaming classes, fields, and methods with obscure names. We applied Proguard to the malware payloads (even when we did not use reflection and encryption) to make the payloads look like real applications.

After a malware payload is created, it must be repackaged into a baseline app. Repackaging malware into a baseline app involves disassembling the app (using `apktool`), and adding information about new components and their interfaces in the application's Manifest file. We then insert code into the Main activity to start the top-level malware dispatcher service (whose activation trigger is configurable), and add malicious code and data files into the apk. We then reassemble the decompiled app using `apktool`. If code insertion has been done correctly, apktool produces a new Android app, which must be signed by `jarsigner` before deployment on a real device.

IV. Real User-driven Execution

Armed with a computationally diverse malware suite, we now select a similarly diverse suite of benign apps, drive them with long, real, user inputs, and extract hardware signals from them. Figure 8 shows the apps that we run along with their inputs – using these, we find that since the apps' traces are so diverse, we need to build HMDs customized for each app.

A. Benign Apps

Our main goal is to choose applications that represent popular usage, and that require permissions to access resources like SD card and internet connectivity. This ensures that the applications are interesting targets for malware. Further, we ensure that the apps cover a mix of compute (games), user driven (medical app, news), and network (radio) behaviors, diversifying the high-level use cases for apps in the benignware suite. Our chosen app set includes native (C/C++/assembly), Android (Dalvik instructions), and web-based functionality, varying the execution environment of our benign app pool. We confirm that this high-level diversity does indeed translate into diverse hardware-level signals.

B. User Inputs

For each benign application, we created a workload that represents common users' behavior according to statistics available online. For example, when exercising Firefox, we visited popular websites listed on alexa.com. Automating this is simple. For Angry Birds, we recorded a user playing the game for multiple rounds and successfully completing several levels. For the medical diagnostics app (Sana), we record users completing several questionnaires, where each questionnaire requires stateful interactions spread over several screens. Such deep exploration of real apps is far beyond the capability of not only the default UI testing tool in Android (Monkey [44]), but also state of the art in input generation research [45]. Without such deep exploration of benign apps, the apps' hardware traces will reflect only a dormant app and cause the malware signals to stand out at test time but not in a deployed system.

For each benign app, we collect 6 user-level sessions (each 5–11 min long) and use a heavily modified Android Reran [46] to record and replay 4 of these sessions with random delays added between recorded actions (while ensuring correct execution of the app). These 10 user-level traces per app generate 56–111 minutes of performance counter traces across all apps.

Each benign app is then repackaged with 66 different payloads to create 9×66 malware samples. To collect performance counter traces, we replay one of the app's user-level traces and extract 5–11 minutes long performance counter traces for *each* malware sample.

Figure 8 shows some interesting trends in benign traces. While Sana commits 145 Billion instructions in 111 minutes, Zombie WorldWar commits 2,261 Billion instructions in 91 minutes – clearly, Sana is much more user-bound while Zombie WorldWar is compute-heavy. CNN and Angry Birds are similar to Zombie WorldWar, where TuneIn Radio lies between Sana and Zombie WorldWar in instructions committed.

Finding: HMDs have to be application-specific. Interestingly, as we show in our evaluation (Section V), the compute intensity of CNN and Zombie WorldWar results in them having the worst detection rates among all the apps in our suite. On

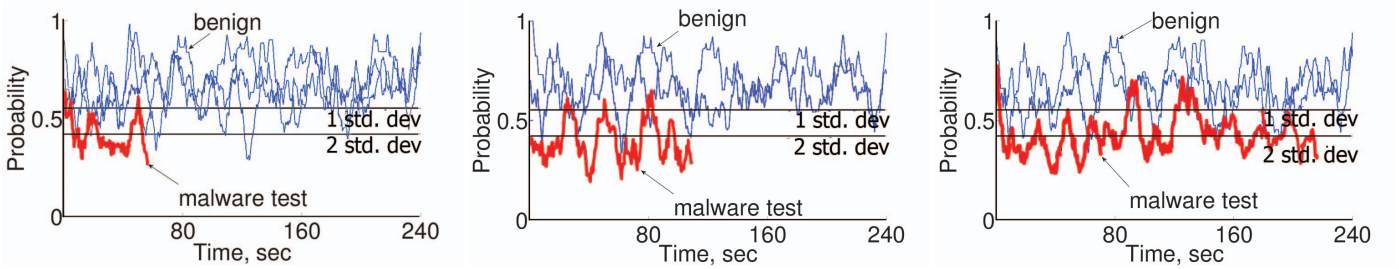

Fig. 9. HMD results for Angry Birds with click fraud operating at three (increasing) intensities. Since HMD is trained on benign AngryBirds, a low dark-line shows that the HMD detects malware as a low probability state.

the other hand, even though TuneIn Radio is more intense than Sana, TuneIn Radio exposes malware better. We find that this is because the Radio has more regular behavior while Sana executes in short, sharp bursts. Sherlock's realistic replay infrastructure and user-input traces are key to producing these insights into HMDs' performance.

C. Extracting Hardware Signals

We now describe our measurement setup for precise reproducibility. The measurement setup requires careful setup and correctness checks since it is difficult to replay real user inputs to the end once delays and malware payloads are added.

Devices. Our experimental setup consists of an Android development board connected to a desktop machine via USB, which in turn stores data on a server for data processing and construction of ML models. The desktop machine uses a wireless router to capture internet traffic generated by the development board. The traffic collected from the router is analyzed to ensure that benignware and malware execute correctly.

We use a Samsung Exynos 5250 equipped with a touch screen, and a TI OMAP 5430 development board, and we reboot the boards between each experiment. We ran all experiments on the Exynos 5250 because some common apps like NYTimes and CNN crashed on OMAP 5430 for lack of a WiFi module, but repeated Angry Birds experiments on the OMAP 5430 to ensure that our results are not an artifact of a specific device.

Performance counter tracing. We used the ARM DS-5 v5.15 framework and the Streamline profiler as a non-intrusive method for observing performance counters. DS-5 Streamline reads data every millisecond and on every context switch, so it can ascribe performance events to individual threads. However, in DS-5 Streamline extracting per process data can only be done using its GUI – we automate this process using the JitBit [47] UI automation tool.

Choice of performance counters. We used hardware performance counters to record five architectural signals: memory loads\stores, immediate and indirect control flow instruction counts, integer computations, and the total number of executed instructions; and one micro-architectural signal: the total number of mispredicted branches. We collected counter information on a per process basis as matching programmer-visible threads to Linux-level threads requires instrumenting the Android middleware (i.e., is non-trivial), and because per-process counters yielded reasonable detection rates. We leave exploring the optimal set of performance counters for future work.

Overhead of counter sampling. We found that sampling counters with 1 ms time resolution incurs less than 0.3% slowdown on the CF-Bench mobile benchmark suite. Prior work also reports low overheads: Demme et al report 5% overhead at 25k cycles per sample, while Tang et al report 1.5% at 512k retired instructions per sample. Beyond sampling, the detection logic itself is fairly simple – while we describe our HMDs in the next section, Ozsoy et. al have shown that a neural net HMD costs less than 5% in area and delay to an AO486 CPU (with overheads expected to be smaller for larger CPUs).

Ensuring correct execution. We ensured that the malicious payload was executed correctly on the board for each trace. Specifically, synthetic malware communicated with a Hercules 3-2-6 TCP server running on the desktop computer, which recorded a log of all communication. The synthetic malware itself printed to a console on the desktop computer (via `adb`) as well as to DS-5 Streamline when running each malicious payload.

For experiments with off-the-shelf malware, we developed an HTTP server to support custom (reverse-engineered) duplex protocols for C&C communication. If we allowed malware to communicate to its original server, which was not under our control, we captured network traffic going through the router. We checked the validity of performance counters readings obtained via DS-5 Streamline with specially crafted C programs, which we compiled and ran natively on the boards.

D. Constructing and Evaluating HMDs

Using benign and malware traces collected as described above, an HMD analyst can then train and test a range of HMD algorithms. For example, Figure 9 shows one of the HMD algorithms we present in a case study in Section V-A. The HMD is an anomaly detector and the figure plots the likelihood that the current trace is going through a known phase—a low probability thus indicates potential malware (the dark line) while higher probabilities indicate benignware (light gray lines). Increasing the payload's intensity lowers the probability even further. By tuning the probability at which a time interval is flagged as malicious (or by training a classifier to learn this), an analyst can trade-off false positives and true positives.

Importantly, we evaluate true positives and the detection threshold using only the time windows that contain malware payload execution. We do *not* use time windows where our

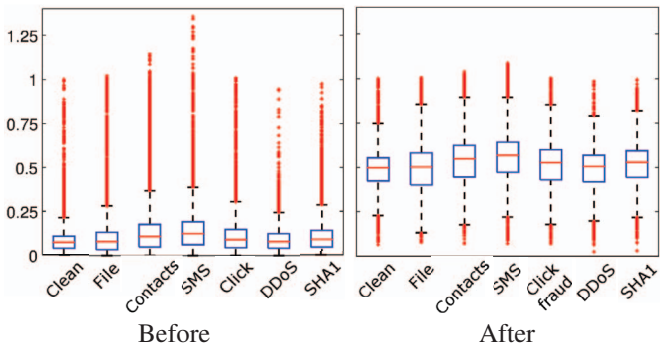

Fig. 10. Distribution of load/store events in Angry Birds before and after power transform. Power transform does not make malware *payloads on Android* more discernible from benign behavior, whereas Tang et al. [6] show that it separates *exploits* from benign apps in Windows.

repackaging code and dispatcher service executes, since we would like the HMD to be evaluated solely using payloads and not exploits. We do not use time windows *before* or *after* the payload is complete, because if an HMD raises an alert when the payload is *not* executing, the alert may in reality be a false positive that will get recorded as a true positive. Prior evaluation methods do not separate out malware payload intervals and may have this error. On the other hand, to measure false positives, we use benign traces only and hence use the entire trace durations for each experiment. Finally, we use 10-fold cross validation on an appropriate subset of our data to evaluate HMDs.

V. Case Studies using Sherlock

We show how malware analysts can use Sherlock through three case studies. **(1)** We use malware payload sizes in Section III to tune the machine learning features (100ms v. sub-ms in prior work) for an unsupervised HMD. Our HMD out-performs prior work designed to detect short-lived exploits by 24.7% on the area under curve (AUC) metric (Section V-A). **(2)** Sherlock's taxonomy of malware in Section III can be used to train a supervised learning based HMD eficiently. This 'balanced' HMD outperforms alternative HMDs – that are trained on subsets of malware behaviors – by 12.5% AUC when tested on new variants of the behaviors. **(3)** Surprisingly, we show that our unsupervised HMD can detect malware that uses obfuscation to evade the best known static analyses. Hence, HMDs and static analyses are complementary and can drive malware payloads towards inefficient implementations.

A. Improving Unsupervised HMDs

We begin by quantifying why prior work designed to detect exploits may not yield the best HMDs to detect long-lived payloads.

Exploit-based ML features do not expose payloads (Figure 10). Tang et al. [6] present an HMD specifically designed to detect the multi-stage exploits that characterize Windows malware. The HMD samples performance counters every 512k cycles, and uses a power transform on performance counter data to separate benign and malicious time intervals. Then, a one-class SVM (ocSVM) is trained on short-lived features – i.e., on each sample as a non-temporal model and using 4 consecutive samples to train a temporal model – to label anomalous time intervals as malicious.

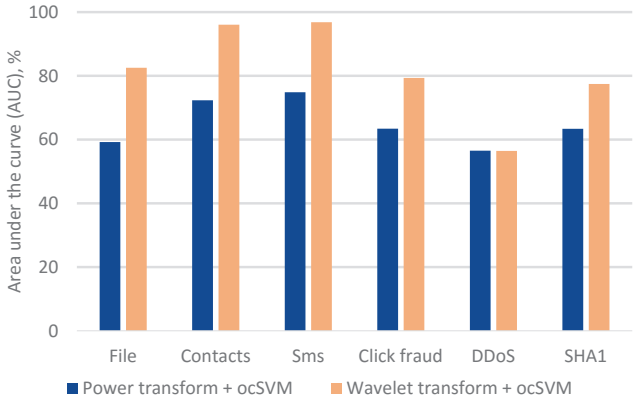

Fig. 11. Comparison of power transform + ocSVM (prior work) and Discrete Wavelet Transform + ocSVM (this work). Our detector has 24.7% better area under curve metric (AUC) than prior work.

We find that power transform does *not* have the same effect on mobile malware payloads—payloads look very similar to benignware traces even after a power transform. For example, Figure 10 shows the distribution of load-store instruction count per time interval for benign Angry Birds (labeled 'Clean'), compared to time intervals in Angry Birds infected with different malware payloads (e.g., file stealer, click fraud, DDoS, etc)—before and after a power transform. The distributions are shown as a box-and-whiskers plot, where the box edges are 25^{th} and 75^{th} percentiles, the central mark is the median, the whiskers extend to the most extreme data points not considered outliers, while the outliers are plotted individually in red. Data in both plots have been normalized to the range of benign Angry Birds' values. We use the standard Box-Cox power transformation to turn performance counter traces into an approximately normal distribution. Since the distributions of malware and benignware in Figure 10 overlap significantly, training an ocSVM on this dataset will yield a poor HMD as we show next.

Payload-centric ML features. We designed a new HMD whose features reflect our findings about mobile malware payload sizes in Figure 7. Specifically, we attempt to capture program effects at the scale of 100ms intervals, i.e., closer to the time required for atomic actions like stealing information or networking activity.

We then extract features from each 100ms long time interval using Discrete Wavelet Transform (DWT) and use the wavelet coefficients as a feature vector for the time interval. The wavelet transform can provide both accurate frequency information at low frequencies and time information at high frequencies, which are important for modeling the execution behavior of the applications. We use a three-level DWT with an order 3 Daubechies wavelet function (db3) to decompose a time interval. We also used the Haar wavelet function with similar detection results.

Finally, we use multiple feature vectors to construct two models: (a) a bag-of-words algorithm followed by a ocSVM, and (b) a probabilistic Markov model. Both these models are simple to train and compute at run-time, and hence serve as good local detectors (and a good baseline for more complex models such as neural nets that are harder to train).

1) Bag-of-words Anomaly Detector: The bag-of-words model treats 100ms time intervals as words and a Time-to-Detection (TTD) window as a document. We experimented with a range of words and TTDs, finding a codebook of 1000 words and TTD = 1.5 seconds to yield good results. The bag of words algorithm maps each TTD window into a 1000-entry histogram, and trains a one-class SVM on benign histograms. We parameterize the one-class SVM so that it has ∼20% percent false positives.

Comparison with power transform — ocSVM HMD. Figure 11 compares our bag-of-words based ocSVM with one that uses a power transform using the area under ROC curve (AUC) metric. Note that AUC is a relative metric to compare classifiers, whereas the operating range measures an HMDs' robustness to atomic-action-sized mutations in malware. The bag-of-words model outperforms prior work for each category of malware behavior and by an average of 24.7% higher AUC across all malware.

Operating range of DWT — bag-of-words — ocSVM. Figure 12 shows the operating range for the bag-of-words model. Each cell in the matrix corresponds to a malware payload action (y-axis) and benign app (x-axis) pair. The malware payloads are grouped by category and within each category, increase in size from top to bottom and in delay from right to left. These experiments use parameters from Figure 8. The intensity of the color – from light green to dark red – corresponds to the detection rate, which is computed as the number of raised alarms versus the total number of alarms that could be raised.

Figure 12 shows that the bag-of-words model achieves, at ∼20% false positive rate: 1) surprisingly high true positive rate for dynamic, compute intensive apps such as Angry Birds (99.9%), CNN (84%), Zombie WorldWar (93%), and Google Translate (92.4%); and 2) ∼80% true positive rate for both Amazon and Sana.

Bag-of-Words model HMD space overheads. Bag-of-Words models require 639KB – 1,229KB space with an average of 840KB and less than 2% of the average size of Android apps.[2]

2) Markov-model based Anomaly Detector: We present an alternative HMD to show that HMD models should be chosen specific to each application, and that there is an opportunity to apply ensemble methods to boost detection rates.

Our first-order Markov model based HMD assumes that the normal execution of an application (approximately) goes through a (limited) number of states (program phases), and the current state depends only on the previous state. The goal is to detect malware if its performance counter trace creates a sequence of rare state transitions (as shown in Figure 9).

The HMD uses DWT to extract features as in the bag-of-words model, but maps them to a smaller number of words (i.e., states in the Markov model) using k-means clustering. We use the Bayesian Information Criterion (BIC) score [48] to find that 10 to 20 states is a good number across all benign apps. Using observed state transitions derived from training data, we empirically estimate the transition matrix and initial probability distribution (through Maximum Likelihood Estimation). For

[2]https://crowdsourcedtesting.com/resources/mobile-app-averages

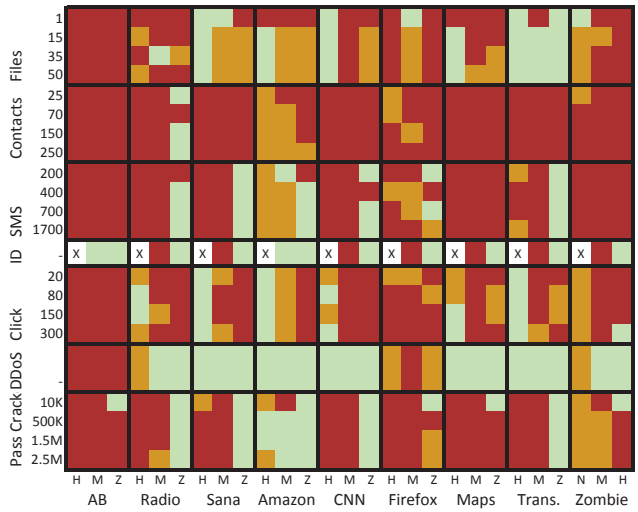

Fig. 12. The operating range of Bag-of-words HMD. In each rectangle, the size of malicious payload grows from the top to the bottom, and the amount of delay decreases from left to right (H=High, M=Medium, Z=Zero delay). If color goes from light to dark within a rectangle, then the detection threshold (i.e., the lower end of the operating range) lies inside the rectangle.

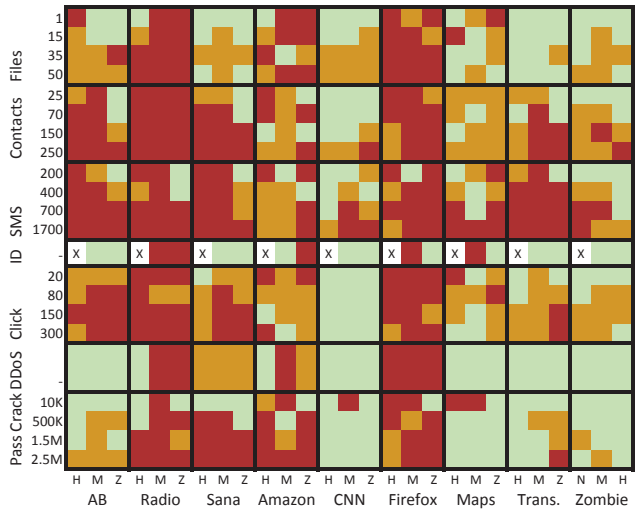

Fig. 13. The operating range of Markov model HMD. Interestingly, the Markov model performs worse than the simpler bag-of-words model for compute intensive and dynamic apps (e.g., Angry Birds, CNN, and Zombie WorldWar).

detection, the Markov model HMD tracks the joint probability of a sequence of states over time and if malware computations create anomalous hardware signals (i.e. this probability is below a threshold for 5 states in our model), the HMD raises an alert.

Operating range of DWT — Markov model HMD. Figure 13 shows the results-matrix for the Markov model based detector. All the results are shown for a false positive rate of 20-25%.

Increasing the size of each payload action makes malware more detectable – this can be seen as the colors being more intense towards the bottom part of most rectangles. Increasing the delay between two malicious actions does not have a similarly predictable effect – SMS stealers in Angry Birds is a rare pair where detection rate increases with delay. This is interesting since intuitively, adding delays between payload actions should decrease the chances of being detected. However, these experiments indicate that for most malware-benign pairs,

detection depends on how each payload action interferes with benign computation rather than delays between the payload actions.

The most important take-away from Figure 13 is that for most malware-benignware pairs, the detectability changes from light green to dark red as we go from top to bottom in the rectangle – this shows that our malware parameters in Figure 13 are close to the detection threshold, i.e. the lower end of the HMD's operating range for the current false positive rate. There are a few exceptions as well, such as click fraud, DDoS, and password crackers hiding in CNN; and DDoS in Angry Birds, Maps, Translate, and Zombie World Wards. For these cases, the payload intensity has to be increased further to find their detection threshold.

Markov model HMD space and time overheads. Markov models representing the behavior of the benign apps vary from 1.2KB to 6.7 KB, with an average size of 3.2KB – they are thus cheap to store on devices and transfer over cellular networks. Its time to detection ranges between 1.2 seconds to 4.4 seconds and about 2.5 seconds on average. This means that the system can detect suspicious activities at the very beginning, considering that exfiltrating even one photo takes 2.86 sec on average.

3) HMDs should be app-specific: Interestingly, the Markov model works significantly better than bag-of-words for TuneIn Radio – with a 10% FP: 90% TP rate compared to 38%FP: 90% TP rate respectively – but performs significantly worse on apps like Angry Birds. In summary, a deployed HMD will benefit from choosing the models that work best for each application, but due to their different TP:FP operating points, will also benefit from using boosting algorithms in machine learning [49].

B. Improving Supervised HMDs

Sherlock can significantly improve performance of supervised learning based HMDs; specifically, by training the HMDs on a 'balanced' training data set that contains malware with each high-level behavior identified in the Section III-B. Note that supervised learning techniques can be trained to recognize specific malware families [5] (i.e. a multi-class model) or to coalesce all malicious feature vectors (FVs) into a single label (i.e., a 2-class model)—we evaluate both categories in Figure 14.

To quantify Sherlock's 'balancing' effect on the resulting performance of a classifier, we conduct the following experiment. We partition the entire malicious set of FVs into a training set and two testing sets such that a training set contains malicious FVs of a particular type (e.g. SMS stealer, file stealer, DDoS attack and etc). The remaining FVs are placed in two non-overlapping testing sets. The first testing set includes the same type of FVs as the training set, while the second one comprises of FVs not in the training set. Finally, we add to each training/testing set benign FVs whose number is equal to the number of malicious FVs in the corresponding set.

Thus, for every partition we conduct two experiments: train a classifier on a training set and test it on the two testing sets. We present the results for Random Forest classifier using ROC curves (left) and AUC metric (right) in Figure 14 to compare relative performance of a classifier under different training and testing data sets. Each ROC curve is labeled as 'training malware type' – 'testing malware type' ('others' means all malware types that are not included in the training set). And the red ROC curve shows the result of training and testing on a 'balanced' malware set.

We show relevant ROC curves in Figure 14 along with the AUC metric for all ROC curves on the right. The light brown bars correspond to AUC of the experiments where we train and test a classifier on the same malware type, i.e. we test it on the first testing set. The blue bars demonstrate classifier's 'cross' performance, i.e. we test it on the rest malware types (on the second testing set). All results are computed using 10-fold cross validation.

We experimented with several supervised learning algorithms – e.g., decision tree, 2-class SVM, k-Nearest Neighbor, Boosted decision trees, and Random Forest (RF)– and present the results for RF classifier because it demonstrated the best performance on our data set.

The common trend that we observed across all nine apps and all malware types is that the RF classifier has significantly better performance when testing on the same malware types (solid lines are higher than the dashed ones). The only exception is when the RF HMD is trained on DDoS malware, it surprisingly achieves better performance on other malware behaviors than on the in-class malware behaviors.

This can be explained by high stealthiness of our implementation of a DDoS attack [50] – each mobile device only opens HTTP connections to a target server and keeps them alive with minimal further requests. Thus, in real apps that also make network connections, DDoS *should* be virtually undetectable using HMDs. From machine learning perspective, DDoS and benign apps are likely closer to each other, while other malware like info-stealers and compute nodes are farther away in feature space, therefore we observe an opposite trend in the case of DDoS experiment in Figure 14.

Further, we trained a classifier on a balanced set of malicious data that included all malware behaviors in Sherlock. The solid line with dots (in the ROC plot) and the column on the far right (in the AUC bar graph) in Figure 14 show that showing some variants of each behavior enables the RF to achieve a higher detection rate (on even new variants) than both prior work as well as one-class SVMs. The RF HMD can, for example, detect close to 85% of the malware with only 5% false positives compared to our anomaly detectors' similar true positives for $\sim$20% false positive rates. Finally, the RF HMD trained on a balanced data set yields 97.5% AUC whereas RF HMDs trained on per-behavior inputs yield AUCs of 91% and 85% when tested against the same or new malware behaviors respectively (averaged across all behaviors).

Operating range of Random Forest HMD. Figure 15 shows the detection results matrix for the RF HMD across the entire malware payload (Y-axis) and benignware (X-axis) categories for a fixed false positive rate of 5%. The key results are that RF detects most payloads except for detecting click fraud and DDoS attacks in CNN, Firefox, and Google Translate. It

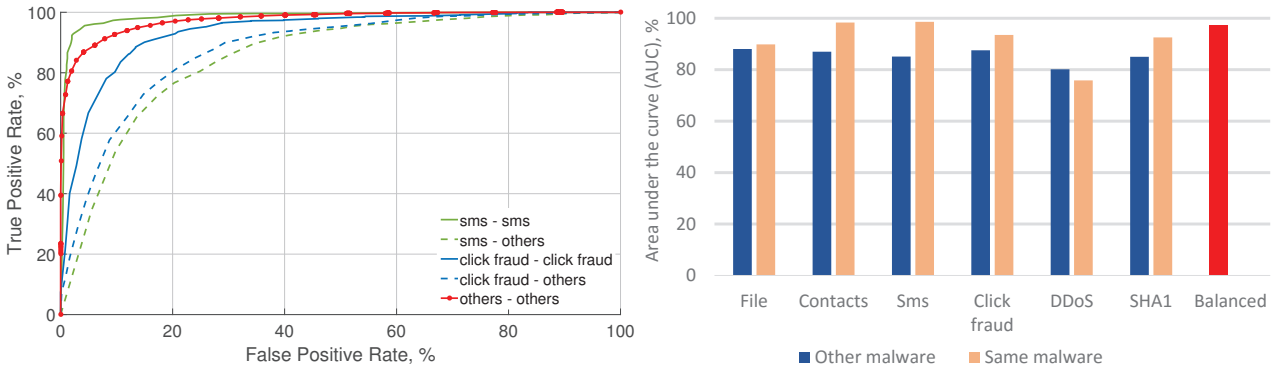

Fig. 14. Training supervised learning HMD on a balanced set of malware behaviors yields best results.

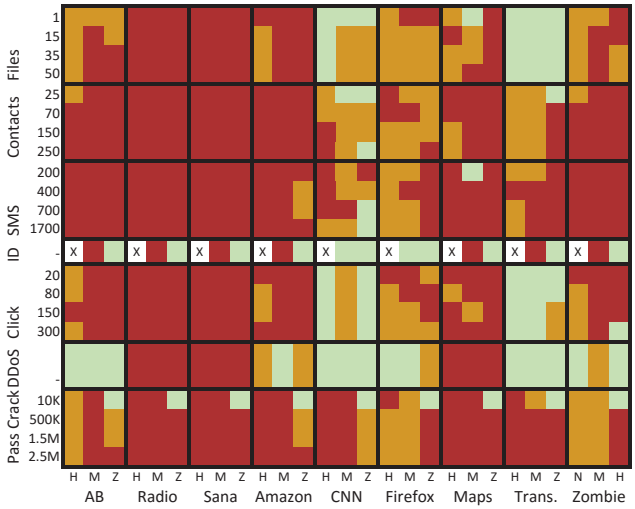

Fig. 15. Operating range of 2-class Random Forest HMD: more effective than anomaly detectors when trained on a balanced dataset of all malware behaviors.

```
1  Code snippet extracted from Obad.apk
2  Method: com.android.system.admin.
3  loOcccoC.loOcccoC(final boolean b)
4
5  dynamically construct class name
6  String class_name = oCIlCll(594, 24, -27);
7  return a class object
8  Class<?> c = Class.forName(class_name);
9  dynamically construct the name of a method
10 String method_name = oCIlCll(250, 33, -51);
11 return an object associated with the method
12 Method m = c.getMethod(method_name,
13           new Class<T>[] { Long.TYPE });
14 m.invoke(value, array);
```

Fig. 16. Code shows Java reflection and string encryption in `Obad` malware that foils static analysis tools.

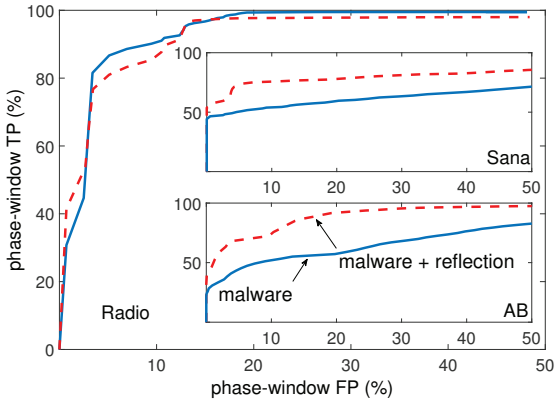

Fig. 17. (Markov model) Effect of obfuscation and encryption on detection rate: interestingly, malware becomes more distinct compared to baseline benign app.

is likely that DDoS attacks – which involve a sequence of infrequent HTTP requests – look very similar to benign apps and are not well suited to be detected using HMDs. Indeed, all three HMDs – bag-of-words, Markov model, and RF – do a poor job of detecting DDoS attacks in most apps. On the other hand, RF consistently detects information stealers and compute malware (password cracker) across most apps. For apps with regular behavior (Radio) or sparse user-driven behavior (Sana), RF can detect all but the smallest of malware payloads.

In summary, Sherlock helps an analyst develop a robust HMD—first by dissecting existing malware to identify orthogonal behaviors, and then by training the HMD on a representative set of malicious behaviors. In the end, using the operating range, Sherlock informs the analyst of the type of behaviors the HMD is well/poorly suited at detecting.

C. Composition with Static Analyses

Reflection is a powerful method for writing malware that evades static program analysis tools used in App Stores today [51]. Interestingly, we show that malware that uses reflection to obfuscate its static program paths in turn worsens its dynamic hardware signals, and improves HMDs' detection rates.

Java methods invoked via reflection are resolved at runtime, making it hard for static code analysis to understand the program's semantics. At the same time, reflection alone is not sufficient – all strings in the code must also be encrypted, otherwise the invoked method or a set of possible methods might be resolved statically.

To illustrate an actual malicious use of Java reflection and encryption, we show a code snippet (Figure 17) from `Obad` malware [29]. The code decrypts class and method names (lines

6 and 10) by calling the method `oCIlCll()`. As a result, static analyses [18], [52] either do not model reflection or conservatively over-approximate the set of instantiated classes for `method_name` (line 10) and target methods for the `invoke` function (line 14). Due to control-flow edges that may never be traversed, static data-flow analysis becomes overly conservative, and static analyses end up with high false positive rates (or more commonly, with malware that goes undetected).

We augmented our synthetic malware with reflection and encryption similar to `Obad`'s implementation. Static analysis of our malware does not reveal any API methods that might raise alarms—we tested this using the Virustotal online service which ran 38 antiviruses on our binary without raising any warnings.

Figure 17 shows results of using the Markov model HMD on the 66 synthetic malware samples from Figure 7 augmented with reflection and encryption, and embedded into each of AngryBirds, Sana, and TuneInRadio. We see that in Angry Birds and Sana the detection rate of the malware that uses both reflection and encryption is significantly higher because reflection and encryption are computationally intensive and disturb the trace of the benign parent app (i.e., more than the same malware without reflection and encryption). We do not see the same trend for TuneInRadio because its detection rate was already quite high, so the additional impact of reflection on TuneIn Radio stays within the noise margin. We conclude that HMDs complement current static analyses and can potentially reduce the pressure on computationally intensive dynamic analyses with a larger trusted code base [17].

VI. Conclusions

HMDs are being studied by processor manufacturers like Qualcomm and Intel. As computer architects explore new hardware signals and accelerators to improve security in general and malware detectors in particular—our work lays a solid methodological foundation for future research into HMDs for mobile platforms. In particular, our approach of identifying *why* a detector succeeds and fails, instead of black-box experiments with malware binaries, is crucial. Indeed, prior work has pointed out the pitfalls of using machine learning in a black-box manner for network-based intrusion detection systems [53]. Our future work will include applying Sherlock's white-box methodology to software detectors and efficiently composing them with HMDs.

Acknowledgment

This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA, NSF awards #1314709 and #1453806, and gifts from Qualcomm and Google.

References

[1] A. P. Felt and D. Wagner, "Phishing on mobile devices," in *In W2SP*, 2011.

[2] "Vulnerable & aggressive adware," http://www.fireeye.com/blog/technical/2013/10/ad-vulna-a-vulnaggressive-vulnerable-aggressive-adware-threatening-millions.html.

[3] "Master key vulnerability," http://blog.trendmicro.com/trendlabs-security-intelligence/trend-micro-solution-for-vulnerability-affecting-nearly-all-android-devices.

[4] N. Peiravian and X. Zhu, "Machine learning for android malware detection using permission and api calls," in *Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, ser. ICTAI '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 300–305. [Online]. Available: http://dx.doi.org/10.1109/ICTAI.2013.53

[5] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 559–570. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485970

[6] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, 2014, pp. 109–129.

[7] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *Proceeding of the 21st International Symposium on High Performance Computer Architecture*, 2015.

[8] K. Khasawneh, M. Ozsoy, C. Donovick, N. Abu-Ghazaleh, and D. Ponomarev, "Ensemble learning for low-level hardware-supported malware detection," in *18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[9] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 361–372. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665726

[10] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," in *BlackHat*, 2015.

[11] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1406–1418. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813708

[12] M. Payer, "Hexpads: A platform to detect "stealth" attacks," in *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, 2016, pp. 138–154. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-30806-7_9

[13] E. Vasilomanolakis, S. Karuppayah, M. Mühlhäuser, and M. Fischer, "Taxonomy and survey of collaborative intrusion detection," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 55:1–55:33, May 2015.

[14] C. V. Zhou, C. Leckie, and S. Karunasekera, "A survey of coordinated attacks and collaborative intrusion detection," *Computers & Security*, vol. 29, no. 1, pp. 124 – 140, 2010.

[15] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers a case study on pdf malware classifiers," in *Network and Distributed Systems Symposium*, 2016.

[16] "Trendlabs a look at google bouncer," http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer.

[17] 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 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, 2010.

[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[19] D. Dash, B. Kveton, J. M. Agosta, E. Schooler, J. Chandrashekar, A. Bachrach, and A. Newman, "When gossip is good: Distributed probabilistic inference for detection of slow network intrusions," in *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2*, ser. AAAI'06. AAAI Press, 2006, pp. 1115–1122.

[20] Y. Xie, H.-A. Kim, D. R. O'Hallaron, M. K. Reiter, and H. Zhang, "Seurat: A pointillist approach to anomaly detection," in *The International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2004.

[21] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malre detection," in *Proceedings of the 2012 International Symposium on Softre Testing and Analysis*, ser. ISSTA 2012. Neork, NY, USA: ACM, 2012, pp. 122–132. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336768

[22] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013,*

Revised Selected Papers, 2013, pp. 86–103. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-04283-1_6
[23] L. Invernizzi, S. Miskovic, R. Torres, C. Kruegel, S. Saha, G. Vigna, S. Lee, and M. Mellia, "Nazca: Detecting malware distribution in large-scale networks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014. [Online]. Available: http://www.internetsociety.org/doc/nazca-detecting-malware-distribution-large-scale-networks
[24] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: ACM, 2013, pp. 11:1–11:1. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488370
[25] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for cpu based attestation and sealing," ser. HASP '13, 2013.
[26] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," in *Proceeding SP '12 Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2016.
[27] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceeding SP '12 Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012, pp. 95–109. [Online]. Available: http://dl.acm.org/citation.cfm?id=2310710
[28] "Mobile malware database," http://contagiominidump.blogspot.com.
[29] "Obad malware," http://securityintelligence.com/diy-android-malware-analysis-taking-apart-obad-part-1.
[30] "Geinimi malware," https://nakedsecurity.sophos.com/2010/12/31/geinimi-android-trojan-horse-discovered/.
[31] "Malware database," http://malware.lu.
[32] "Malware database," http://virusshare.com.
[33] "Universal android rooting procedure (rage method)," http://theunlockr.com/ 2010/10/26/universal-android-rooting-procedure-rage-method/.
[34] "Gingerbreak apk root," http://droidmodderx.com/ gingerbreak-apk-root-your-gingerbread-device.
[35] "Exploid," http://forum.xda-developers.com/showthread. php?t=739874.
[36] E. Chin and D. Wagner, "Bifocals: Analyzing webview vulnerabilities in android applications," in *Revised Selected Papers of the 14th International Workshop on Information Security Applications - Volume 8267*, ser. WISA 2013. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 138–159.
[37] "Evernote patches," http://blog.trendmicro.com/trendlabs-security-intelligence/evernote-patches-vulnerability-in-android-app/.
[38] "Applock vulnerability," http://blog.trendmicro.com/trendlabs-security-intelligence/applock-vulnerability-leaves-configuration-files-open-for-exploit.
[39] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046779
[40] "Android rat malware," http://www.itpro.co.uk/malware/22627/android-rat-malware-invades-mobile-banking-apps.
[41] "Mobile bitcoin miner," https://blog.lookout.com/blog/2014/04/24/badlepricon-bitcoin.
[42] M. Kazdagli, L. Huang, V. Reddi, and M. Tiwari, "Morpheus: Benchmarking computational diversity in mobile malware," in *Workshop on Hardware and Architectural Support for Security and Privacy*, 2014.
[43] "http://developer.android.com/tools/help/proguard.html." [Online]. Available: http://developer.android.com/tools/help/proguard.html
[44] "Ui/application exerciser monkey," http://developer.android.com/tools/help/monkey.html.
[45] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450
[46] "Record and replay for android," http://www.androidreran.com.
[47] "Jitbit macro recorder," http://www.jitbit.com/.
[48] D. Pelleg and A. W. Moore, "X-means: Extending k-means with efficient estimation of the number of clusters," in *Proceedings of the 7th International Conference on Machine Learning*, 2000.
[49] R. Schapire and Y. Freund, *Boosting: Foundations and Algorithms*. MIT Press, 2012.
[50] "Slow loris attack," http://www.slashroot.in/slowloris-http-dosdenial-serviceattack-and-prevention.
[51] "Dissecting android's bouncer," https://www.duosecurity.com/blog/dissecting-androids-bouncer.
[52] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382223
[53] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *In Proceedings of the IEEE Symposium on Security and Privacy*, 2010.