On the Feasibility of Online Malware Detection with Performance Counters

John Demme Matthew Maycock Jared Schmitz Adrian Tang
Adam Waksman Simha Sethumadhavan Salvatore Stolfo
Department of Computer Science, Columbia University, NY, NY 10027
jdd@cs.columbia.edu, mhm2159@columbia.edu, {jared,atang,waksman,simha,sal}@cs.columbia.edu

ABSTRACT

The proliferation of computers in any domain is followed by the proliferation of malware in that domain. Systems, including the latest mobile platforms, are laden with viruses, rootkits, spyware, adware and other classes of malware. Despite the existence of anti-virus software, malware threats persist and are growing as there exist a myriad of ways to subvert anti-virus (AV) software. In fact, attackers today exploit bugs in the AV software to break into systems.

In this paper, we examine the feasibility of building a malware detector in hardware using existing performance counters. We find that data from performance counters can be used to identify malware and that our detection techniques are robust to minor variations in malware programs. As a result, after examining a small set of variations within a family of malware on Android ARM and Intel Linux platforms, we can detect many variations within that family. Further, our proposed hardware modifications allow the malware detector to run securely beneath the system software, thus setting the stage for AV implementations that are simpler and less buggy than software AV. Combined, the robustness and security of hardware AV techniques have the potential to advance state-of-the-art online malware detection.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/software interfaces*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Invasive software*

General Terms

Security in Hardware, Malware and its Mitigation

Keywords

Malware detection, machine learning, performance counters

[1]This work was supported by grants FA 99500910389 (AFOSR), FA 865011C7190 (DARPA), FA 87501020253 (DARPA), CCF/TC 1054844 (NSF), Alfred P. Sloan fellowship, and gifts from Microsoft Research, WindRiver Corp, Xilinx and Synopsys Inc. Any opinions, findings, conclusions and recommendations do not reflect the views of the US Government or commercial entities.

ISCA '13 Tel-Aviv, Israel

1. INTRODUCTION

Malware – short for malicious software – is everywhere. In various forms for a variety of incentives, malware exists on desktop PCs, server systems and even mobile devices like smart phones and tablets. Some malware litter devices with unwanted advertisements, creating ad revenue for the malware creator. Others can dial and text so-called "premium" services resulting in extra phone bill charges. Some other malware is even more insidious, hiding itself (via rootkits or background processes) and collecting private data like GPS location or confidential documents.

This scourge of malware persists despite the existence of many forms of protection software, antivirus (AV) software being the best example. Although AV software decreases the threat of malware, it has some failings. First, because the AV system is itself software, it is vulnerable to attack. Bugs or oversights in the AV software or underlying system software (*e.g.,* the operating system or hypervisor) can be exploited to disable AV protection. Second, production AV software typically use static characteristics of malware such as suspicious strings of instructions in the binary to detect threats. Unfortunately, it is quite easy for malware writers to produce many different code variants that are functionally equivalent, both manually and automatically, thus defeating static analysis easily. For instance, one malware family in our data set, AnserverBot, had 187 code variations. Alternatives to static AV scanning require extremely sophisticated dynamic analysis, often at the cost of significant overhead.

Given the shortcomings of static analysis via software implementations, we propose hardware modifications to support secure efficient dynamic analysis of programs to detect malware. This approach potentially solves both problems. First, by executing AV protection in secure hardware (with minimum reliance on system software), we significantly reduce the possibility of malware subverting the protection mechanisms. Second, we posit that dynamic analysis makes detection of new, undiscovered malware variants easier. The intuition is as follows: we assume that all malware within a certain family of malware, regardless of the code variant, attempts to do similar things. For instance, they may all pop up ads, or they may all take GPS readings. As a result, we would expect them to work through a similar set of program phases, which tend to exhibit similar detectable properties in the form of performance data (*e.g.,* IPC, cache behavior).

In this paper, we pose and answer the following central feasibility question: *Can dynamic performance data be used to characterize and detect malware?* We collect longitudinal, fine-grained microarchitectural traces of recent mobile

Android malware and Linux rootkits on ARM and Intel platforms respectively. We then apply standard machine learning classification algorithms such as KNN or Decision Trees to detect variants of known malware. Our results indicate that relatively simple classification algorithms can detect malware at nearly 90% accuracy with 3% false positives for some mobile malware.

We also describe hardware support necessary to enable online malware detection in hardware. We propose methods and techniques for (1) collection of fine-grained runtime data without slowing down applications, (2) secure execution of AV algorithms to detect at runtime the execution of malware and (3) secure updating of the AV algorithms to prevent subversion of the protection scheme.

Another important contribution of the paper is to describe the experimental framework for research in the new area of hardware malware detection. Towards this we provide a dataset used in this research. This dataset can be downloaded from: `http://castl.cs.columbia.edu/colmalset`.

The rest of the paper is organized as follows. In Section 2 we provide background on malware, then describe the key intuition behind our approach (Section 3), and explain our experimental method (Section 4), followed by evaluations of Android ARM malware, x86 Linux Rootkits and side channels (Sections 5, 6, 7). We describe hardware support in Section 8 and our conclusions in Section 9.

2. BACKGROUND ON MALWARE

In this section, we provide an abbreviated and fairly informal introduction on malware.

2.1 What is Malware and Who Creates It?

Malware is software created by an attacker to compromise security of a system or privacy of a victim. A list of different types of malware is listed in Table 1. Initially created to attain notoriety or for fun, malware development today is mostly motivated by financial gains [1, 2]. There are reports of active underground markets for personal information, credit cards, logins into sensitive machines in the United States, etc. [3]. Also, government-funded agencies (allegedly) have created sophisticated malware that target specific computers for espionage or sabotage [4, 5, 6]. Malware can be delivered in a number of ways. To list a few, an unsuspecting user can be tricked into: clicking on links in "phishing" emails that download and install malware, opening email attachments with malicious pdfs or document files, browsing web pages with exploits, using infected USB sticks or downloading illegitimate applications repackaged to appear as normal applications through mobile stores.

2.2 Commercial Malware Protections

The most common protection against malware is anti-virus (AV) software. Despite what the name anti-virus suggests, anti-virus can also detect and possibly remove categories of malware besides viruses. A typical AV system works by scanning files during load time for known signatures, typically code strings, of malware. Figure 1 shows how anti-virus signatures are prepared: Honeypots collect malware and non-malware which are then analyzed by humans to create signatures. These signatures are then delivered to the host anti-virus software periodically.

A complementary approach to signature-based detection is also used in practice [7]. In reputation based AV detection, users anonymously send cryptographic signatures of executables to the AV vendor. The AV vendor then determines how often an executable occurs in a large population of its users to predict if an executable is malware: often, uncommon executable signatures occurring in small numbers are tagged as malware. This system is reported to be effective against polymorphic and metamorphic viruses but does not work against non-executable threats such as malicious pdfs and doc files [8]. Further it requires users to reveal programs installed on their machine to the AV vendor and trust the AV vendor not to share this secret.

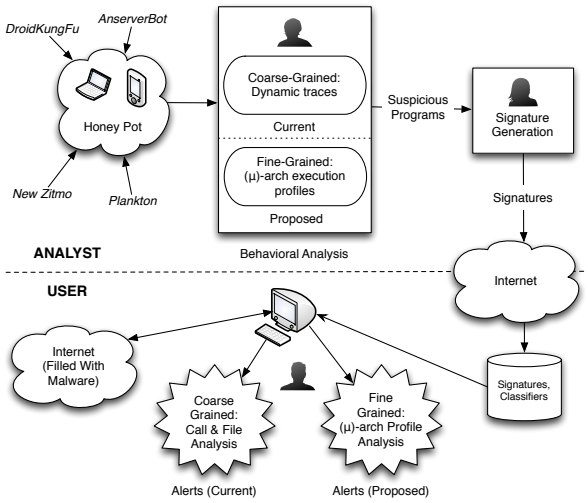

Figure 1: AV signature creation and deployment.

2.3 How Good is Anti-Virus Software?

Just like any other large piece of software, AV systems tend to have bugs that are easily exploited, and thus AV

Table 1: Categories of Malware

Malware	Brief Description
Worm	Malware that propagates itself from one infected host to other hosts via exploits in the OS interfaces typically the system-call interface.
Virus	Malware that attaches itself to running programs and spreads itself through users' interactions with various systems.
Polymorphic Virus	A virus that, when replicating to attach to a new target, alters its payload to evade detection, *i.e.* takes on a different shape but performs the same function.
Metamorphic Virus	A virus that, when replicating to attach to a new target, alters both the payload and functionality, including the framework for generating future changes.
Trojan	Malware that masquerades as non-malware and acts maliciously once installed (opening backdoors, interfering with system behavior, etc).
AdWare	Malware that forces the user to deal with unwanted advertisements.
SpyWare	Malware that secretly observes and reports on users computer usage and personal information accessible therein.
Botnet	Malware that employs a user's computer as a member of a network of infected computers controlled by a central malicious agency.
Rootkit	Malware that hides its existence from other applications and users. Often used to mask the activity of other malicious software.

protections are easily bypassed. In a recent paper, Jana and Shmatikov [9] found that all of the 36 commercially available AV systems they examined could be bypassed. Specifically, they detected many bugs in the code that parse program binaries which either allowed bad code to pass undetected or gain higher privilege. They argued that the problem of building robust parsers (and hence software malware detectors) is not easy since the number of file formats is quite large, and many of their specifications are incomplete in several ways. Their paper demonstrates the futility in trying to secure complex, million-line softwares like AV. Unlike software detectors, the hardware malware detectors we propose do not have to deal with multiple executable formats. Instead they work on single input format – integer streams from performance counters. Further, they are not easily turned off. Thus hardware detectors are significant step towards more robust detectors.

2.4 Malware Arms Race

There is an arms race between malware creators and detectors. The earliest detectors simply scanned executables for strings of known bad instructions. To evade these detectors, attackers started encrypting their payloads. The detectors, in response, started scanning for the decryption code (which could not be encrypted) packed with the malware. The malware creators then started randomly mutating the body of the payload by using different compilation strategies (such as choosing different register assignments or padding NOPs) to create variants [10].

In response to these advances in malware creation, defenders were motivated to consider *behavioral* detection of malware instead of static signatures. Behavior-based detection characterizes how the malware interacts with the system: what files it uses, the IPC, system call patterns, function calls and memory footprint changes [11, 12, 13]. Using these characteristics, detectors build models of normal and abnormal program behaviors, and detect abnormal execution by comparing against pre-built behavioral models. Many of these schemes use machine learning techniques to learn and classify good and bad behaviors from labeled sets [14, 15, 16, 17].

2.5 Improving Malware Detection

While behavioral schemes permit richer specification of good and bad behaviors than static checkers, they tend to have high performance overheads since the more effective ones demand creation and processing of control- and data-flow graphs. Because of their overheads behavior-based detectors are not typically used on end hosts, but analysts in malware-detection companies may use them to understand malware-like behaviors. All of these techniques are envisioned to be implemented in software.

In this work, for the first time, we use hardware performance counters for behavior based detection of malware, and describe the architecture necessary to support malware detection in hardware. Our performance counter based technique is a low-overhead technique that will not only allow analysts to catch bad code more quickly, it may also be feasible to deploy our system on end hosts. Unlike static signature based detection AV, we aim to detect variants of malware from known malware signatures. Unlike reputation based system our scheme does not require users to reveal programs installed on their computer.

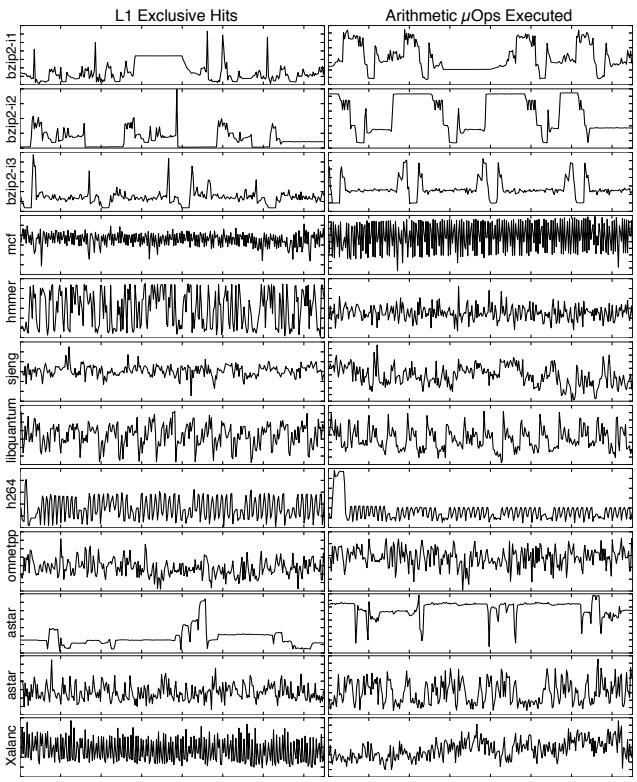

Figure 2: Performance counter measurements over time in the SPEC benchmark suite. We also observe readily apparent visual differences between the applications. Intuitively, we expect it to be possible to identify programs based on these data.

Recent research has also examined using hardware performance counters for detecting anomalous program behaviors [18, 19]. This is a different and (intuitively) harder problem than attempted here. The anomaly detection works aim to detect small deviations in program behavior during an attack such as a buffer overflow or control flow deviation from otherwise mostly benign execution. In contrast, we attempt to identify execution of whole programs such as key logger when it is run, typically as the end result of exploitation such as buffer overflow vulnerability.

3. KEY INTUITION

A major thesis of this paper is that runtime behavior captured using performance counters can be used to identify malware and that the minor variations in malware that are typically used to foil signature AV software do not significantly interfere with our detection method.

The intuition for this hypothesis comes from research in program phases [20, 21]. We know that programs exhibit phase behavior. They will do one activity A for a while, then switch to activity B, then to activity C. We also know that programs tend to repeat these phases – perhaps the program alternates between activities B and C. Finally, and most importantly, it has been shown that these phases correspond to patterns in architectural and microarchitectural events.

Another important property of program phases and their behaviors is that they differ radically between programs. Figure 2 plots event counts over time for several SPEC appli-

cations. In it, we see the differences between the benchmarks as well as interesting phase behavior. Given these data, it seems intuitive that these programs could be differentiated based on these time-varying signatures.

Our hypothesis that minor variations in malware do not significantly affect these data cannot be inferred from previous work. Rather, it is based on two observations:

• First, regardless of *how* malware writers change their software, its semantics do not change significantly. For instance, if a piece of malware is designed to collect and log GPS data, then no matter how its writer re-arranges the code, it still collects and logs GPS data.

• Second, we assume that in accomplishing a particular task there exist subtasks that cannot be radically modified. For instance, a GPS logger will always have to warm up the GPS, wait for signals, decode the data, log it and at some future point exfiltrate the data out of the system. As a result of these invariant tasks, we would expect particular phases of the malware's execution to remain relatively invariant amongst variations.

Since it is not obvious that either of our arguments are true, we quantitatively test them in subsequent sections. In summary, we find that when we build detection methods using a small subset of variations, these detection methods often work on the other variations as well.

4. EXPERIMENTAL SETUP

Can simple performance metrics be used to identify malware? To answer this question we conduct several feasibility studies. In each, we collect performance counter data on malware and train a set of classifiers to detect malicious behavior. In addition to data from malware programs, we collect data from non-malware programs (Figure 3). Here we describe our program sets, provide details of our data collection infrastructure, describe our classifiers, and discuss types and granularity of malware detection.

4.1 Malware & Non-Malware Programs Used

In this study we used 503 malware and 210 non-malware programs from both Android ARM and Intel X86 platforms. The full list of programs is available in the dataset website[1]. The malware programs were obtained from three sources. First from the authors of previous work studying Android malware [22], and second from a website[2] that contains a large number of malware. We also obtained two publicly available Linux x86 rootkits [23, 24]. Data from non-malware programs serve two purposes: during training as negative examples, and during testing to determine false positive rates, *i.e.,* the rate of misclassifying non-malware.

For the purposes of this paper, we use a wide definition of malware. Malware is any part of any application (an Android APK file or rootkit binary) that has been labeled as malware by a security analyst. We use this definition to enable experimentation with a large amount of malware, which is necessary for supervised machine learning.

This definition of malware is, however, inaccurate. Much malware comes attached to legitimate code, so users often execute malware alongside their desired applications. As such, an accurate definition would require malware samples

[1] `http://castl.cs.columbia.edu/colmalset`
[2] `http://contagiominidump.blogspot.com/`

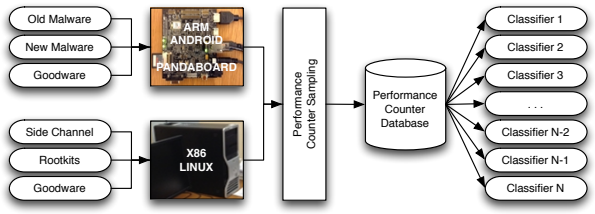

Figure 3: Our workflow for malware experiments.

that have undergone deep forensic analysis to determine exact portions that result in malicious actions, and to identify inputs or environmental conditions under which the malware actually performs malicious actions.

As researchers designing a problem for supervised machine learning algorithms, this presents a particular challenge: what parts of our "malware" data should be labeled as such for training? Should we label the entirety of the software as malicious while much of our "malicious" training data could actually be mostly benign? The only other option is to laboriously pick out the good threads or program portions from the bad. This latter option, however, is neither scalable nor practical and to the best of our knowledge not available even in datasets from commercial vendors [25].

While our definition of malware is inaccurate, it is practical. However, it makes our classification task more difficult since our classifiers see both malicious behaviors and legitimate behaviors with "malware" labels during training. With more accurate labels we would likely see lower false positive rates and higher malware identification rates. In other words, our experimental framework is conservative and one would expect better results in practice. Our experiments are only designed only to demonstrate feasibility.

4.2 Data Collection

We have written a Linux kernel module that interrupts once every N cycles and samples *all* of the event counters along with the process identifier of the currently executing program. Using this tool we collect multi-dimensional time-series traces of applications like those shown in Figure 2.

Our data collection tool is implemented on two platforms. For x86 workloads, we run Linux 2.6.32 on an 8 core (across two sockets) Intel Xeon X5550 PC with TurboBoost up to 2.67GHz and 24GB of memory. These processors are based on Intel's Nehalem design, which implement four configurable performance counters, so our Intel x86 data is 4-dimensional. For ARM workloads, we run Android 4.1.1-1 which is based on Linux 3.2. We use a distribution of Android from Linaro that runs on Texas Instrument's PandaBoard, a demonstration board for their OMAP4460 processor with dual ARM Cortex-A9 cores. ARM architectures of this generation have six configurable performance counters, so ARM data is 6-dimensional. To mitigate performance overheads from interrupts, we use sampling periods of 50,000 and 25,000 cycles for Intel and ARM respectively.

Bias Mitigation: We aim to mimic real-world deployment conditions as much as possible when collecting data. There are a variety of factors that could affect our results: (1) Contamination – malware does its best to infect machines and be persistent, possibly influencing subsequent data captures. We control for this by wiping and restoring all non-volatile storage in between data captures for different malware families and, more importantly, between data

collection runs of the training and testing set. (2) Environmental noise and input bias: these two factors cannot be controlled in deployment conditions, so in order to make our problem both more difficult and realistic, we do not control for them. (3) Network connectivity: some malware requires an internet connection, so our test systems were connected over Ethernet and were not firewalled or controlled in any way, as they would be in the wild. (4) User bias: We had three different users collect data in arbitrary order for the training and testing runs to mitigate systematic biases in interacting with applications. (5) Ensuring successful malware deployment: We cannot say with certainty if malware actually worked during a run. While the consequences were clear for some malware such as adware, for some malware we observed unexplainable behaviors, such as the system crashing. It is unknown to us whether these bizarre behaviors were intended or not (there are no specification documents for malware), so all data collected was included, possibly polluting our training and testing data, again likely making our classification task more difficult.

4.3 Machine Learning Methods

In machine learning, classifiers are able to examine data items to determine to which of N groups (classes) each item belongs. Often, classification algorithms will produce a vector of probabilities which represent the likelihoods of the data item belonging to each class. In the case of malware detection, we can simply define two classes: malware and non-malware. As a result, the output from each of out classifiers will be two probabilities representing the likelihood of the data item being malware.

Features Our data collection produces multi-dimensional time series data. Each sample is a vector made up of event counts at the time of sampling. In addition to that, we can also aggregate multiple samples, and then use the aggregate to build feature vectors. Aggregation can even out noise to produce better trained classifiers or dissipate key signals depending on the level of aggregation and the program behavior. In this paper, we experiment with a number of different feature vectors: (1) raw samples (2) aggregations all the samples between context swaps using averages or sums, (3) aggregations between context swaps (previous option) with a new dimension that includes the number of samples aggregated in a scheduling quanta, (4) histograms in intervals of execution. This last one, histograms, breaks up the samples into intervals of fixed size (32 or 128 samples) and computes discrete histograms (with 8 or 16 bins) for each counter. It then concatenates the histograms to create large feature vectors (192 or 768 on ARM).

Classifiers There are a large number of classifiers we could use. Classifiers broadly break down into two classes: linear and non-linear. Linear algorithms attempt to separate n-dimensional data points by a hyperplane – points on one side of the plane are of class X and points on the other side of class Y. Non-linear classifiers, however, have no such restrictions; any operation to derive a classification can be applied. Unfortunately, this means that the amount of computation to classify a data point can be very high. In choosing classifiers to implement for this paper we choose to focus on non-linear algorithms as we did not expect our data to be linearly separable. Here we briefly describe the algorithms we implement:

- In k-Nearest Neighbors (KNN), the classifier is trained by inserting the training data points along with their labels into a spatial data structure like a kd-tree. In order to classify a data point, that point's k nearest neighbors (in Euclidean space) are found using the spatial data structure. The probability that the data point is of each class is determined by how many of its neighbors are of that class and their Euclidean distance.
- Another way to classify data points is to use a decision tree. This tree is built by recursively splitting training data into groups on a particular dimension. The dimension and split points are chosen to minimize the entropy with each group. These decisions can also integrate some randomness, decreasing the quality of the tree but helping to prevent over training. After some minimum entropy is met or a maximum depth is reached, a branch terminates, storing in the node the mix of labels in its group. To classify a new data point, the decision tree is traversed to find the new point's group (leaf node) and return the stored mix.
- One way to increase the accuracy of a classifier is to use multiple different classifiers and combine their results. In a random forest, several (or many) decision trees are built using some randomness. When classifying a new data point, the results of all trees in the forest are weighted equally.
- Finally we attempt classification with Artificial Neural Networks (ANNs). In our neural nets, we define one input neuron for each dimension and two output nodes: one for the probability that malware is running, and one for the probability that non-malware is running. We train our ANNs using backpropagation.

For implementation, we use KNN, Decision Trees, and Random Forests from the Waffles ML library[3]. For our ANNs, we use the FANN library[4].

4.4 Training and Testing Data

As mentioned before many production malware detectors build blacklists using static malware signatures. As a result, they can only detect malware that the AV vendor has already discovered and cataloged. Minor variations thereof – which are relatively easy for attackers to produce – cannot be detected in the wild using existing signatures. If we wanted to, we could design a hardware detector that works exactly as the software signature AV. We would evaluate the feasibility of this by running the same malware multiple times under different conditions to produce the training and testing data. But in this work we want to design a more robust malware detector that in addition to detecting known malware, will also detect new variants of known malware. In order evaluate this functionality, we train a classifier on data from one set of programs – non-malware and variants of malware in a family. We then test the classifier's accuracy on different variants of malware in the same family (and also on non-malware programs). To mitigate bias, the data for training and testing are collected in separate runs without knowledge of whether the data is to be used for testing or training. The data is also collected by different users.

4.5 Classification Granularity

Our data collection can procure performance counter data every 25,000 or 50,000 cycles with little slowdown. So in theory we can classify malware at the granularity of each

[3]waffles.sourceforge.net 2012-08-31
[4]fann.sourceforge.net FANN-2.2.0

sample. However, due to large number of small variations in programs we should expect a large number of false positives. We have indeed found this to be the case, and in fact, we obtained high false positives even at a coarser granularity of every operating system context swap. Due to space considerations we do not present these results. As such, in this paper, we present classification results for malware at two even coarser granularities: thread and application group. In the thread based classification, each thread is classified as malware (or non-malware) by aggregating the classification probabilities for all data points in that thread. In application group level classification, we classify Android Java packages and package families as malware. This approach requires our classifier to determine if, for example, "com.google.chrome" is malicious or not and allows the classifier to use samples from any thread executing that code.

5. DETECTING ANDROID MALWARE

With the rise of Android has come the rise of Android malware. Although Android has the concept of permissions, permissions-based approach often fail because users typically provide permissions indiscriminately or can be tricked into giving permissions by the application. For instance, a fake application packaged like Skype can trick the user into giving permissions to access the camera and microphone. In fact, several Android malware mask themselves as legitimate software, and it is not uncommon for malware writers to steal existing software and repackage it with additional, malicious software.

5.1 Experimental Design

The Android malware data sets are divided up into families of variants. In families with only one variant, we use the same malware but different executions of it for training and testing. For families with more than one variant, we statically divide them up, using $\frac{1}{3}$ for training and the rest for testing. The names of each family and the number of installers (APKs) for each can be found in our results, Table 2. In total our data set includes nearly 368M performance counters samples of malware and non-malware.

Classifier Parameters The classification algorithms outlined in Section 4 can be parameterized in different ways. For instance, for k-Nearest Neighbors, k is a parameter. We search a large space of classifiers, varying many parameters. In order to determine the best set of parameters, we want to choose the classifier that identifies the most malware correctly. However, as we make the classifier more sensitive, we find more malware but also identify some legitimate software as malware. In order to determine which classifier to use, we find the one that performs best (on the training data) for a given false positive percentage. As a result, the results we present are not necessarily monotonically increasing.

Re-training Optimization Since malware applications are known to include both malicious and benign code, we use an optimization to select data points for training that are more likely to be from the malicious part of an malware package. We first train our classifier on all data points. We then run all of our training data through this classifier and sort the data based on the classifier's score, *i.e.,* we calculate the probability of data being malware as called by the malware classifier. We then use only the most "malware-like" data in our training set to re-train the classifier, which we then use in evaluation. The intuition behind this technique

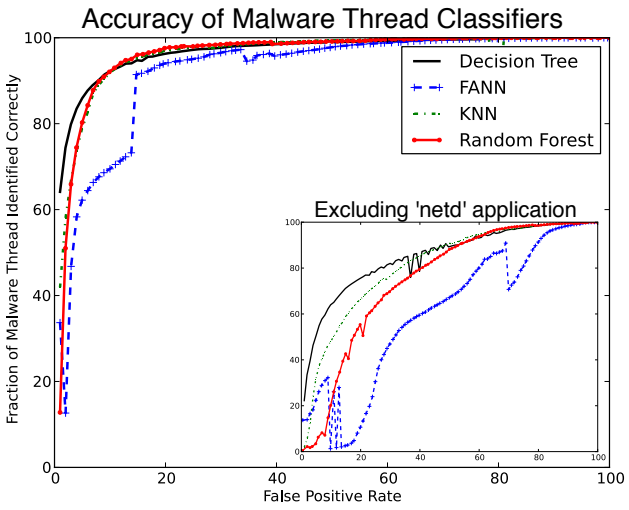

Figure 4: The accuracy of binary classifiers in determining whether or not each running thread is malware.

is that the non-malicious parts of our training data are likely to look a lot like non-malware to the classifier, so we use our initial classifier to filter out those data. In many cases, this retraining allows us to retrain with a smaller amount of data while achieving comparable accuracy (and speeding up training and testing.) In the case of decision trees, we find that this technique significantly improves results. Further, it creates relatively small decision trees, so the computational requirements of classifying each sample is orders of magnitude lower than some of the other methods.

Next we report results on detecting malware at the granularity of threads and at the application level.

5.2 Malware Thread Detection Results

Testing The classification metric we use is the percentage of threads correctly classified. For instance if the malware application has T threads, our classifier, in the ideal case, will flag only those subset of threads that perform malicious actions. For non-malware, ideally all threads should be flagged as benign. As mentioned before, the testing data samples are obtained from a separate run from training and under different input and environmental conditions. We also use different non-malware applications in testing than in training to ensure that we do not build a *de facto* white- or blacklist of applications.

Training We ensure that an equal number of samples from malware and non-malware are used for training. Strictly speaking this is unnecessary but we did it to prevent our classifier results from being biased by the volume of samples from the two categories. The samples are chosen without any relation to the number of threads to mitigate classification bias due to thread selection.

Results Figure 4 shows malware detection by thread in a form similar to a typical ROC curve. As expected, if we allow some false positives, the classifiers find more malware. These results indicate that performance counter data can, with simple analysis, be used to detect malware with relatively good accuracy. Further analysis of results shows that a single application makes up the majority of non-malware during the testing phase. This application is an Android sys-

tem application called "netd" and is responsible for dynamically reconfiguring the system's network stack. As such, it runs often, and our classifiers are excellent at correctly predicting this application as non-malware. If we remove this application from our testing data, we obtain the results inlaid in Figure 4. While they are not as good, they remain positive. We further break down our results by malware family in Table 2. This table shows the number of APKs we were able to obtain for each family along with the number of threads observed. It also shows the number of threads that our classifier correctly identified while maintaining a 10% or better false positive rate. We find a range of results, depending on the family.

Table 2: Malware Families for Training and Testing

Malware Family	APKs	Training Threads	Testing Threads	Threads Flagged	Rate
Tapsnake	1	31	3	3	100%
Zitmo	1	5	1	1	100%
Loozfon-android	1	25	7	7	100%
Android.Steek	3	9	9	9	100%
Android.Trojan. Qicsomos	1	12	12	12	100%
CruseWin	1	2	4	4	100%
Jifake	1	7	5	5	100%
AnserverBot	187	9716	11904	11505	96.6%
Gone60	9	33	67	59	88.1%
YZHC	1	9	8	7	87.5%
FakePlayer	6	7	15	13	86.7%
LoveTrap	1	5	7	6	85.7%
Bgserv	9	119	177	151	85.3%
KMIN	40	43	30	25	83.3%
DroidDreamLight	46	181	101	83	82.2%
HippoSMS	4	127	28	23	82.1%
Dropdialerab	1	18*	16*	13	81.3%
Zsone	12	44	78	63	80.8%
Endofday	1	11	10	8	80.0%
AngryBirds-LeNa.C	1	40*	24*	19	79.2%
jSMSHider	16	101	89	70	78.7%
Plankton	25	231	551	432	78.4%
PJAPPS	16	124	174	136	78.2%
Android.Sumzand	1	8	9	7	77.8%
RogueSPPush	9	236	237	184	77.6%
FakeNetflix	1	27	8	6	75.0%
GEINIMI	28	189	203	154	75.9%
SndApps	10	110	77	56	72.7%
GoldDream	47	1160	237	169	71.3%
CoinPirate	1	8	10	7	70.0%
BASEBRIDGE	1	14*	72	46	63.8%
DougaLeaker.A	6	12*	35*	22	62.9%
NewZitmo	1	5	8	5	62.5%
BeanBot	8	122	93	56	60.2%
GGTracker	1	16	15	9	60.0%
FakeAngry	1	7	10	5	50.0%
DogWars	1	14	8	2	25.0%

* Indicates that data collectors noticed little activity upon launching one or more of the malware APKs, so we are less confident that the payload was successfully achieved.

5.3 Malware Package Detection Results

Testing For application/package-based malware detection, our classifiers use samples from all the threads belonging to a particular software. For instance, all of the samples collected from the testing set of Anserverbot are used to determine whether or not that set of software is malware.

Training In the previous experiment on detecting malware granularity by threads, we used an equal number of samples for both malware and non-malware, but did not normalize the number of samples by application or malware family. In this study, however, in addition to using an equal number of samples for non-malware and malware, we use an equal number of samples from each malware family and an equal number of samples from each non-malware application. This ensures that during training our classifiers see data from any application that ran for a non-trivial amount of time and they are not biased by application run times during the training phase. Since we want to have equal number of samples, we leave out short-running applications and malware families that produce fewer than 1,000 samples.

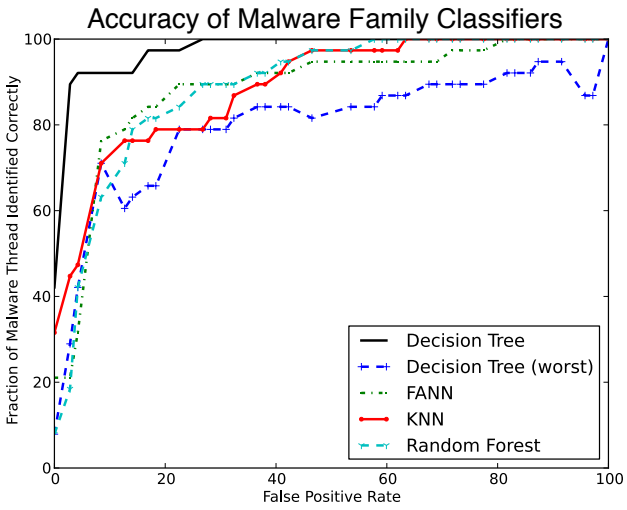

Figure 5: The accuracy of binary classifiers in determining whether families of malware and normal Android packages are malware.

Results The results of our package classifiers are found in Table 3 and Figure 5. The results are equally positive by application as they are for threads. As in the last experiment (thread results), we ran a large number of classifiers with different parameters and selected the best parameter for each false positive rate based on the accuracy of the classifier on the training data (these were 100s of different classifiers). However, unlike the last study, we found that our decision tree classifiers did near-perfectly on all the training data so we could not pick one best parameter configuration. In Figure 5 we show the best and worst accuracies we obtained with different parameters for the decision trees which performed near-perfectly on the testing data. Future work should consider a methodology for selecting classifier parameters in such cases of ties. The table shows raw classifier scores for our malware and some non-malware, both sorted by score. The particular classifier results showcased here aggregate raw decision tree scores from all samples collected from each malware family and non-malware package. We see that on average our malware scores are higher for malware than non-malware. There is, however, some overlap, creating some false positives.

5.4 Conclusions on Android Malware

In our experiments we are testing on a different set of variants from those we train on, showing that our classifiers would likely detect new malware variants in the field that security investigators had not yet seen. Table 4 shows the area under the curve for both schemes with 10% false positives. This is a capability that static signature-based virus

Table 3: Malicious Package Detection Results: Raw Scores

Score	Malware Family	Score	Malware Family	Score	Malware Family	Score	Goodware
0.67	YZHC	0.61	CruseWin	0.58	NewZitmo	0.55	appinventor.ai_todoprogramar.
0.66	Tapsnake	0.61	BASEBRIDGE	0.58	DogWars		HappyWheelsUSA
0.65	Android.Sumzand	0.61	Bgserv	0.57	GEINIMI	0.53	com.android.keychain
0.65	PJAPPS	0.61	DougaLeaker.A	0.56	FakePlayer	0.53	com.pandora.android
0.64	Loozfon-android	0.61	jSMSHider	0.56	AngryBirds-LeNa.C	0.51	com.bestcoolfungames.antsmasher
0.63	SndApps	0.61	FakeAngry	0.55	Android.Trojan.Qicsomos		....
0.63	GGTracker	0.61	Jifake	0.53	GoldDream	0.38	com.twitter.android
0.62	Gone60	0.61	RogueSPPush	0.53	RogueLemon	0.38	com.android.packageinstaller
0.62	FakeNetflix	0.60	Android.Steek	0.53	AnserverBot	0.37	com.android.inputmethod.latin
0.62	Zsone	0.60	Dropdialerab	0.49	Plankton	0.36	android.process.media
0.62	CoinPirate	0.60	HippoSMS	0.49	BeanBot	0.44	Average
0.62	Zitmo	0.60	Endofday	0.47	LoveTrap		
0.61	DroidDreamLight	0.59	KMIN	0.59	Average		

Table 4: AUC below 10% False Positive Rates

Classifier	Thread Detection	Package Detection
Decision Tree	82.3	83.1
KNN	73.3	50.0
Random Forest	68.9	35.7
FANN	53.3	38.4

scanners lack, so there is little basis for comparison. We also showed that our results are consistently positive for two different detection granularities (and thus metrics) increasing our confidence in our malware detection scheme.

Are these results as good as they can be? We are unable to answer this question. The reason is that malware often includes both malicious and non-malicious code, but we do not attempt to separate them. As a result, we label all the threads in malware APKs as malicious in our testing set. But what if only half the threads are responsible for malicious behavior whereas the other half are legitimate code which was not present in our training data? Were this the case, it could well be that we are perfectly detecting all the malicious threads.

Nonetheless, many of our results are quite promising. For instance, after training on data from only five of our YZHC variants, the remaining variants are given significantly higher malware scores than our unseen non-malware. Similarly, after training on only $\frac{1}{3}$ of AnserverBot's variants, threads from the remaining variants are tagged as malware far more often than are non-malware. With further refinement in terms of labeling data and better machine learning methods, we expect that accuracy could be improved significantly.

6. DETECTING LINUX ROOTKITS

Rootkits are malicious software that attackers install to evade detection and maximize their period of access on compromised systems. Once installed, rootkits hide their presence, typically by modifying portions of the operating systems to obscure specific processes, network ports, files, directories and session log-on traces. Although there exist open-source tools like *chkrootkit*[5] and *rkhunter*[6] to detect rootkits, their use of known signatures makes it easy for rootkits to evade detection by varying their behaviors. Furthermore, since these tools work on the same software level as the rootkits, they can be subverted.

[5] `http://www.chkrootkit.org/`

[6] `http://rkhunter.sourceforge.net/`

6.1 Experimental Design

In this case study, we examine the feasibility of rootkit detection with performance data. We examine the two publicly available Linux rootkits which give an attacker the ability to hide log-on session traces, network ports, processes, files and directories. The Average Coder Rootkit works as a loadable kernel module that hides traces via hooking the kernel file system function calls [23]. The Jynx2 Rootkit functions as a shared library and is installed by configuring the LDPRELOAD environment variable to reference this rootkit [24].

To exercise these rootkits, we run the "ps", "ls", "who", and "netstat" Linux commands and monitor their execution. The Average Coder rootkit is used to hide processes, user logins and network connections whereas the Jynx2 rootkit affects "ls" to hide files. To introduce some input bias and collect multiple samples for both training and testing, we run each command with a variety of different arguments. We run half the commands before the rootkit is installed and half after. After data collection, we split the executions up into training and testing sets. Since we do not repeat commands with the same arguments, our training data are input biased *differently* from our testing data, making the learning task both more difficult and more realistic. To increase the variability in our data, we also simulate various user actions like logging in/out, creating files, running programs and initiating network connections. Lastly, to protect against contamination, we wiped our system between installation of the rootkits and collection of "clean" data.

For this case study, we also show the results from an additional classifier: tensor density. This classifier discretizes the vector space into many buckets. Each bucket contains the relative density of classes in the training data set. A data point is classified by finding its bin and returning the stored mix. Although simple, the tensor has $O(1)$ lookup time, so the classifier is very time-efficient.

6.2 Results

We experimented with five different classifiers, the results of which are presented in Figure 6. The "combined" classifier was trained and tested on all of the above programs whereas the other experiments used data from only one of the programs.

Our rootkit identification results are interesting, though not quite as good as the results presented for Android malware in Section 5. The reason rootkit identification is extremely difficult is that rootkits do not operate as independent programs. Rather, they dynamically intercept pro-

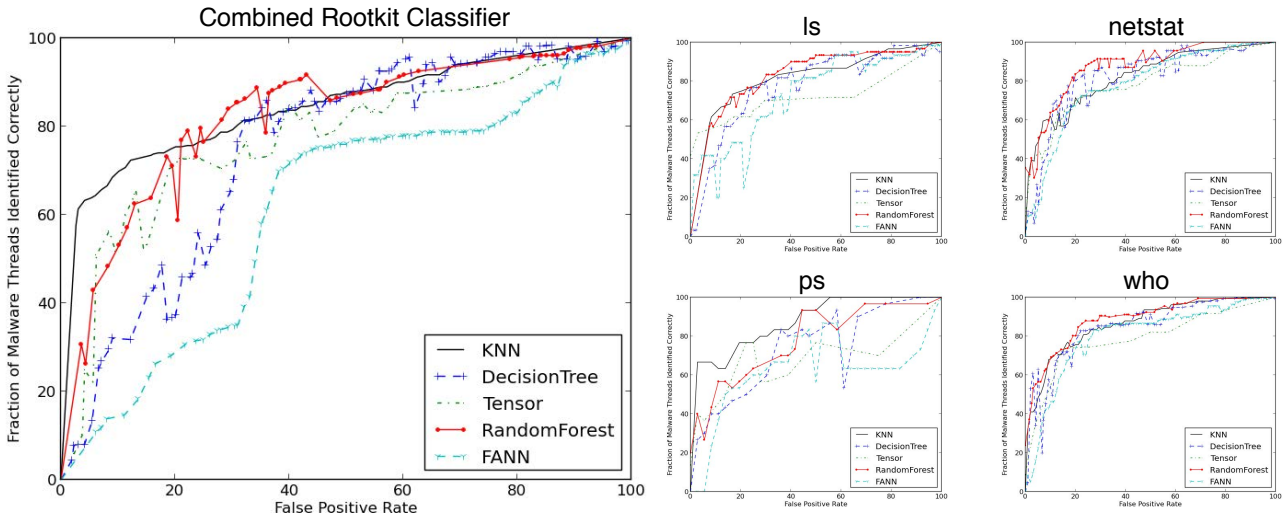

Figure 6: Accuracy of rootkit classifiers on several applications in addition to a classifiers trained and test on all of the applications combined.

grams' normal control flows. As a result, the data we collect for training is affected only slightly by the presence of rootkits. Given these difficulties, we believe our rootkit detection shows promise but will require more advanced classification schemes and better labeling of the data to identify the precise dynamic sections of execution that are affected.

7. SIDE-CHANNEL ATTACKS

As a final case study, we look at side-channel attacks. Side-channel attacks are not considered malware. However, they also threaten security, and we find that our methods can be used even to detect these attacks.

In a side-channel attack unintentional leaks from a program are used to infer program secrets. For example, cryptographic keys can be stolen by observing the performance of the branch predictor or of the caches for many microprocessor implementations. Nearly any system is vulnerable to side-channel attacks [26].

In a microarchitectural side-channel attack, a *victim* process is a process that has secrets to protect and an *attacker* process attempts to place itself within the system in such a way that it shares microarchitectural resources with the victim. Then it creates interference with the victim, *e.g.,* thrashes a shared resource constantly so as to learn the activity of the victim process with respect to that shared resource. The interference pattern is then mined to infer secrets. Since the attackers' interference pattern is programmed we intuitively expect that attacker programs that exploit microarchitectural side-channels should have clear signatures.

Experimental Design To test our intuition we examine one very popular class of side-channel attacks known as a cache side-channel attack. We hypothesize that one particular method for this type of attack - the prime and probe attack method - is a good target for hardware anti-virus. To test our hypothesis, we implement several variants of the standard prime-and-probe technique. In this technique, an attacker program writes to every line in the L1 data cache. The program then scans the cache repeatedly — using a pattern chosen an compile time — reading every line. Whenever a miss occurs, it means there was a conflict miss caused by the victim process sharing the cache. The result of a successful prime-and-probe attack is data about the cache lines used by the victim process over time. Using OpenSSL as the victim process, we compare cache side-channel attack processes against a wide array of benign processes. These benign programs include SPEC2006 int, SPEC2006 fp, PARSEC, web browsers, games, graphics editors and other common desktop applications, as well as generic system-level processes.

Results We train our machine learning algorithms on one third of our total data: 3872 normal program threads and 12 attack threads. We then test our classifiers on the other $\frac{2}{3}$ of the data. Our results in this case are perfect. We catch 100% of the attackers and do not have any false positives on all four classifiers we used. These results demonstrate that cache side-channel attacks are easy to detect with performance counters. We have tested a sub-type of side-channel attacks on one microarchitectural structure but it is likely that other types of microarchitectural side-channel attacks are also detectable. While these initial results are promising further study is necessary to prove this hypothesis.

8. HARDWARE SUPPORT

Moving security protection to the hardware level solves several problems and provides some interesting opportunities. First, we can ensure that the security system cannot be disabled by software, even if the kernel is compromised. Second, since the security system runs beneath the system software, it might be able to protect against kernel exploits and other attacks against hypervisors. Third, since we are modifying the hardware, we can add arbitrary static *and* dynamic monitoring capabilities. This gives the security system unprecedented views into software behavior.

The overall hardware security system that we propose is shown in Figure 7. The system has four primary components:

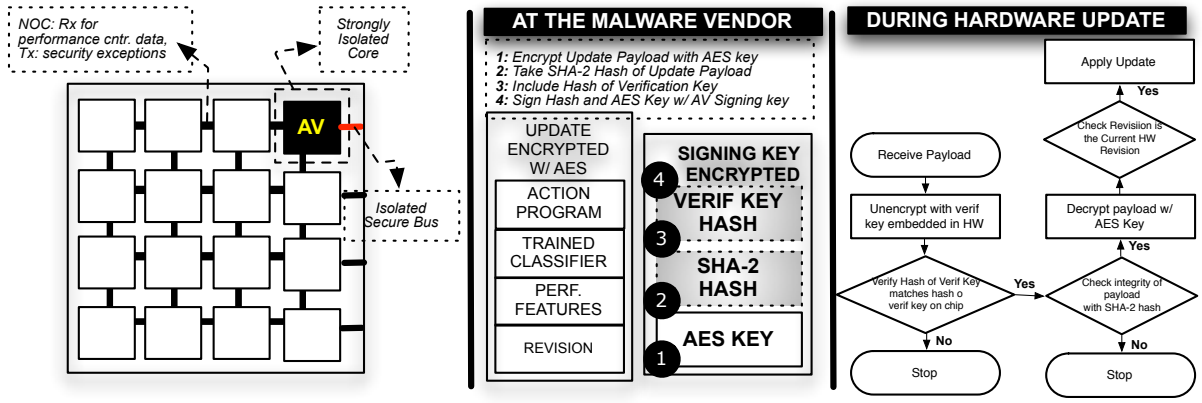

Figure 7: Hardware Architecture for AV Execution and Update Methods for Performance Counter Based AV

Data Collection We must define what data the security processor can collect and how that data is collected and stored.

Data Analysis The security system must analyze incoming data to determine whether or not malicious behavior is occurring.

Action System If a threat is detected by the system, it must react in some manner. This may involve measures as extreme as shutting down the system or as mild as reporting the threat to a user.

Secure Updates Any security measure must, from time to time, be updated to deal with the latest malware. However, these updates must be secure to ensure that only a trusted authority can update the system.

There are many ways to implement a hardware malware detector. The most flexible solution is to allocate one or more general-purpose cores which allows any classification algorithm to be used for detection. Alternatives include microcontrollers or microcontrollers with special-purpose malware detection units that are located on chip, on-chip/off-chip FPGA, and off-chip ASIC co-processor. These choices represent different trade-offs in terms of flexibility and area- and energy-efficiency that need to be explored in detail in the future. In the rest of the section, however, we focus on the backbone system framework required to realize any of these design choices. As we discuss the system framework, we make recommendations or highlight research advancements needed to enable online malware detection with performance counters.

But first, a note on terminology: irrespective of the design choice *i.e.,* microcontroller, accelerator, big or little cores, on-chip unit or off-chip co-processor, FPGA or ASIC, we refer to the entity hosting the classifier algorithm as the AV engine and the units running the monitored programs as targets.

8.1 System Architecture

The system architecture should allow the AV engine: (1) to run independently of any operating system or the hypervisor, and at the highest privilege level in the system. This is to enable continuous monitoring of software at all levels in the stack (2) to enable access to physical memory to store classifier data and (3) to provide strong memory and execution isolation for itself. Isolation ensures that the AV engine is not susceptible to denial-of-service attacks due to resource provisioning (*e.g.,* memory under- or over-flow), or resource contention (*e.g.,* stalling indefinitely due to excessive congestion on the network-on-chip).

Some of these features already exist in processor architectures today. For instance, AMD processors allow a core to carve out a region of the physical memory and lock down that physical memory region from access by other cores [27]. Similarly, some architectures support off-chip coprocessors to have dedicated and isolated access to physical memory through IOMMUs. These features must be extended with mechanisms that guarantee starvation-freedom in shared resources such as the memory controller and in the network-on-chip (or buses in the case of an off-chip AV) to ensure robust communication between the AV engine and the targets.

Recommendation #1 Provide strong isolation mechanisms to enable anti-virus software to execute without interference.

8.2 Data Collection

From the perspective of implementing an A/V engine in hardware, no additional information beyond performance information and thread ID is necessary for thread-based classification. For application-level classification, hardware will need application level identifiers associated with each thread. Thread IDs can already be obtained by hardware.

The AV engine receives performance counter information periodically from the targets. In our experiments, the data from the performance counters is fetched once every 25,000 cycles. This translates to a bandwidth requirement of approximately a few hundred KB/s per target. If the number of active targets (which is at most cores-times-simultaneous-threads many) is not too large like in today's systems, we can design off-chip AV engines using simple serial protocols (such as I2C) with round-robin collection of data from targets. However, as the number of cores increases, on-chip solutions will become more relevant.

Performance data can be either pulled or pushed from the targets. In the pull model – the model used in our experiments – the targets are interrupted during execution to read their performance counters which impacts performance

(roughly 5% empirically). If future hardware support allows performance counters to be queried without interruption, these overheads can be reduced to effectively zero. Another modification that would simplify the design of the AV engine would be to set up the counters to push the data periodically to the AV engine.

The amount of storage required to store the ML data varies greatly depending on the type of classifier used for analysis. For the KNN algorithm, the data storage was roughly 50 MB for binary classification. On the other hand, other analyses needed only about 2.5 MB. Given the variability in storage size and the amount needed, it appears that AV engines will most certainly need mechanisms to access physical memory for retrieving stored signatures.

Recommendation #2 Investigate both on-chip and off-chip solutions for the AV implementations.

Recommendation #3 Allow performance counters to be read without interrupting the executing process.

Recommendation #4 Ensure that the AV engine can access physical memory safely.

8.3 Data Analysis

A wide variety of classifiers can be implemented for data analysis. In this paper we experiment with four well-known classifiers to estimate their potential for malware identification. Most likely, advances in machine learning algorithms and implementations will enable better classification in the future. To allow for this flexibility it appears that general purpose cores are preferable to custom accelerators for the AV engine. However, the AV engine may present domain-specific opportunities for instruction customization, such as special types of memory instructions or microarchitectural innovations in terms of memory prefetchers.

A classification scheme is at best as good as the discerning power of its features. We show that current performance counters offer a good number of features that lead to good classification of malware. However, it is likely that the accuracy can be improved further if we included more features. Thus, we add our voice to the growing number of performance researchers requesting more performance counter data in commercial implementations. Specifically, from the point of view of our malware detection techniques, information regarding instruction mixes and basic block profiles for regions would be very helpful. These inputs can inform the analysis of working-set changes.

Recommendation #5 Investigate domain-specific optimizations for the AV engine.

Recommendation #6 Increase performance counter coverage and the number of counters available.

8.4 Action System

Many security policies can be implemented by the AV engine. Some viable security policies are:

• *Using the AV engine as a first-stage malware predictor.* When the AV engine suspects a program to be malicious it can run more sophisticated behavioral analysis on the program. Hardware analysis happens 'at speed' and is orders of magnitude faster than behavioral analysis used by malware analysts to create signatures. Such pre-filtering can avoid costly behavioral processing for non-malware programs.

• *Migrating sensitive computation.* In multi-tenant settings such as public clouds, when the AV engine suspects that an active thread on the system is being attacked (say through a side-channel) then the AV engine can move the sensitive computation. Of course, in some scenarios it may be acceptable for the AV system to simply kill a suspect process.

• *Using the AV engine for forensics.* Logging data for forensics is expensive as it often involves logging all interactions between the suspect process and the environment. To mitigate these overheads, the information necessary for forensics can be logged only when the AV engine suspects that a process is being attacked.

Thus there are a broad spectrum of actions that can be taken based on AV output. The AV engine must be flexible enough to implement these security policies. Conceptually, this means that the AV engine should be able to interrupt computation on any given core and run the policy payload on that machine. This calls for the AV engine to be able to issue a non-maskable inter-processor interrupt. Optionally, the AV engine can communicate to the OS or supervisory software that it has detected a suspect process so that the system can start migrating other co-resident sensitive computation.

Recommendation #7 The AV engine should be flexible enough to enforce a wide range of security policies.

Recommendation #8 Create mechanisms to allow the AV engine to run in the highest privilege mode.

8.5 Secure Updates

The AV engine needs to be updated with new malware signatures as they become available or when new classification techniques are discovered. The AV update should be constructed in a way to prevent attackers from compromising the AV. For instance, a malicious user should not be able to mute the AV or subvert the AV system to create a persistent, high-privilege rootkit.

We envision that each update will contain one or more classifiers, an action program that specifies security policies, a configuration file that determines which performance features are to used with what classifiers, and an update revision number. This data can be delivered to the AV engine securely using techniques used for software signing but requires a few tweaks to allow it to work in a hardware setting. The process is described in Figure 7.

First, we require that the AV engine implements the aforementioned flowchart directly in hardware: this is because we do not want to trust any software, since all software is potentially vulnerable to attacks. Second, we require hardware to maintain a counter that contains the revision number of the last update and is incremented on every update. This is to prevent an attacker from rolling back the AV system, which an attacker might do to prevent the system from discovering new malware. The AV engine offers this protection by rejecting updates from any revision that is older than the revision number is the hardware counter. In other words, there are fast-forwards but no rewinds.

Recommendation #9 Provide support in the AV engine for secure updates.

9. CONCLUSIONS

In this paper we investigate if malware can be detected in hardware using data available through existing performance counters. If possible, it would be a significant advance in the area of malware detection and analysis, enabling malware detection with very low overheads. Further, it would allow us to build malware detectors which are invisible to

the system, in the hardware beneath the operating system.

The intuition that drove us to ask this question was the observation that programs appear to be unique in terms of their time-series behavior, while variants of the same programs do similar things. Our results indicate that this intuition is true. We can often detect small changes to running programs (rootkit detection) or be somewhat insensitive to variations (malware detection) depending on how we train our classifier.

We demonstrate the feasibility of our detection methods and highlight the increased security from leveraging hardware, but more research is necessary. First, our detector accuracy can be improved. This will involve further research into classification algorithms and ways to label malware data more accurately. Second, our classifiers are not optimized for hardware implementations. Further hardware/algorithm co-design can increase accuracy and efficiencies.

Despite our results it is not clear if dynamic analysis like ours provides a significant advantage to defenders in the malware arms race. While we are able to detect some variants, could virus writers simply continue permuting their malware until it evades our detector? Would this level of change to the virus require some human intervention, making the task more difficult? We suspect that our techniques increases difficulty for virus writers. This is because the virus writer now needs to take into account a wide range of microarchitectural and environmental diversity to evade detection. This is likely difficult, thus the bar for repeatable exploitations is likely to be higher. However, this topic merits further study.

Traditionally, the problem of dealing with malware has been relegated to the software community. Although much work has been done in the area, the limitations of pure software approaches in practical settings are significant. The addition of hardware support takes the fight against malware to a new arena, one with different and perhaps higher standards for attack compared to the state-of-the-art.

Acknowledgements

The authors thank anonymous reviewers, Prof. Martha Kim and members of the Computer Architecture and Security Technologies Lab (CASTL) at Columbia University for their feedback on this work. They also thank Computing Resource Facilities at Columbia University Department of Computer Science for their technical assistance in server management.

10. REFERENCES

[1] B. Stone-Gross, R. Abman, R. Kemmerer, C. Kruegel, D. Steigerwald, and G. Vigna, "The underground economy of fake antivirus software," in *Economics of Information Security and Privacy III* (B. Schneier, ed.), pp. 55–78, Springer New York, 2013.

[2] J. Caballero, C. Grier, C. Kreibich, and V. Paxson, "Measuring Pay-per-Install: The commoditization of malware distribution," in *Proc. of the 20th USENIX Security Symp.*, 2011.

[3] Trend Micro Corporation, "Russian underground."

[4] R. Langner, "Stuxnet: Dissecting a Cyberwarfare Weapon," *Security & Privacy, IEEE*, vol. 9, no. 3, pp. 49–51, 2011.

[5] Laboratory of Cryptography and System Security (CrySyS Lab), "sKyWIper: A Complex Malware for Targeted Attacks," Tech. Rep. v1.05, Budapest University of Technology and Economics, May 2012.

[6] E. Chien, L. OMurchu, and N. Falliere, "W32.Duqu: The Precursor to the Next Stuxnet," in *Proc. of the 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2012.

[7] Z. Ramzan, V. Seshadri, and C. Nachenberg, "Reputation-based security: An analysis of real world effectiveness," Sep 2009.

[8] L. Bilge and T. Dumitras, "Before we knew it: an empirical study of zero-day attacks in the real world," in *Proc. of the 2012 ACM conf. on Computer and communications security*, pp. 833–844, 2012.

[9] S. Jana and V. Shmatikov, "Abusing file processing in malware detectors for fun and profit," in *IEEE Symposium on Security and Privacy*, pp. 80–94, 2012.

[10] P. SzÃűr and P. Ferrie, "Hunting for metamorphic," in *In Virus Bulletin Conference*, pp. 123–144, 2001.

[11] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "Accessminer: using system-centric models for malware protection," in *Proc. of the 17th ACM conf. on Computer and communications security*, pp. 399–412, 2010.

[12] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proc. of the the 6th joint meeting of the European software engineering conf. and the ACM SIGSOFT symp. on The foundations of software engineering*, ESEC-FSE '07, pp. 5–14, 2007.

[13] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proc. of the 1996 IEEE Symp. on Security and Privacy*, pp. 120–135, 1996.

[14] W. Lee, S. J. Stolfo, and K. W. Mok, "A data mining framework for building intrusion detection models," in *In IEEE Symposium on Security and Privacy*, pp. 120–132, 1999.

[15] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Proc. of the 5th intl. conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 108–125, Springer-Verlag, 2008.

[16] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *Proc. of the 10th intl. conf. on Recent advances in intrusion detection*, RAID'07, pp. 178–197, Springer-Verlag, 2007.

[17] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Network and Distributed System Security Symposium*, 2009.

[18] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," in *Proc. of the sixth ACM workshop on Scalable trusted computing*, pp. 71–76, 2011.

[19] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *Proc. of the 2012 42nd Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*, pp. 1–12, 2012.

[20] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *Micro, IEEE*, vol. 23, pp. 84 – 93, nov.-dec. 2003.

[21] C. Isci, G. Contreras, and M. Martonosi, "Live, runtime phase monitoring and prediction on real systems with application too dynamic power management," in *Proc. of the 39th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, pp. 359–370, 2006.

[22] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symp. on*, pp. 95 –109, may 2012.

[23] F. Matias, "Linux rootkit implementation," Dec 2011.

[24] BlackHat Library, "Jynx rootkit2.0," Mar 2012.

[25] T. Dumitras and D. Shou, "Toward a standard benchmark for computer security research: the worldwide intelligence network environment (wine)," in *Proc. of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, pp. 89–96, ACM, 2011.

[26] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-Channel Vulnerability Factor: A Metric for Measuring Information Leakage," in *The 39th Intl. Symp. on Computer Architecture*, pp. 106–117, 2012.

[27] A. M. Azab, P. Ning, and X. Zhang, "Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proc. of the 18th ACM conf. on Computer and communications security*, (New York, NY, USA), pp. 375–388, ACM, 2011.