An Anomaly Detection Fabric for Clouds Based on Collaborative VM Communities

Rashid Tahir, Matthew Caesar
University of Illinois Urbana-Champaign

Ali Raza, Mazhar Naqvi, Fareed Zaffar
Lahore University of Management Sciences

***Abstract*—The vast attack surface of clouds presents a challenge in deploying scalable and effective defenses. Traditional security mechanisms, which work from inside the VM fail to provide strong protection as attackers can bypass them easily. The only available option is to provide security from the layer below the VM i.e., the hypervisor. Previous works that attempt to secure VMs from "outside" either incur substantial space or compute overheads making them slow and impractical or require modifications to the OS or the application codebase. To address these issues, we propose an anomaly detection fabric for clouds based on system call monitoring, which compresses the stream of system calls at their source making the system scalable and near real-time. Our system requires no modifications to the guest OS or the application making it ideal for the data center setting. Additionally, for robust and early detection of threats, we leverage the notion of *VM/container communities* that share information about attacks in their early stages to provide immunity to the entire deployment. We make certain aspects of the system flexible so that vendors can tune metrics to offer customized protection to clients based on their workload types. Detailed evaluation on a prototype implementation on KVM substantiates our claims.**

I. Introduction

The sheer volume and immense size of modern day clouds, makes them hard to protect and consequently, vulnerable to abuse. With a large perimeter and an intricate interior (numerous middleboxes), clouds have a sizable attack surface. Traditional defense mechanisms simply cannot scale to cover the entire expanse and must be spread thin to protect against more frequent vulnerabilities and exploits. As a result, hackers have found numerous "entry points" to break into clouds, using either network or host-based infiltration vectors, causing considerable monetary losses to both vendors and users. From digital currency mining pools [4], [5] to spam relays [1] and from unlimited storage banks [38] to illegal file sharing applications [50], hackers are exploiting clouds to great effect.

The situation is further exacerbated by the fact that these break-ins often go completely unnoticed [3], [4], [2] as defenses, such as antiviruses, are mostly deployed inside the VM or container instance. In the case of VM hijacking, such mitigation mechanisms are easy to bypass or turn off if the attacker gets root access. Alternatively, in the case of malicious deployments by attackers, using compromised user accounts [5] or free resources [4], defense mechanisms are altogether ignored as the attacker himself is the "admin". To prove the fact that vendors are unaware of these break-ins, researchers recently managed to setup a large botnet on Amazon EC2 that did not get flagged by any detection or mitigation mechanism that EC2 might have had [4]. Given these challenges and the stakes involved, vendors need to deploy scalable defenses, "outside" of the containers/VMs.

As far as the cloud network is concerned, vendors have resorted to hardware-based solutions, such as Anti-DDoS Boxes [8], Intrusion Detection and Prevention Systems [9], Vulnerability Scanners [11] etc. These middleboxes, though very costly to deploy and even more expensive to maintain, offer a certain degree of protection to the edge as well as the core against common threats as hardware can handle load much better than software and alleviates some of the scale issues for the vendor. However, the real problem, where hardware-based solutions are either non-existent or offer very limited protection, is the cloud host forcing vendors to primarily rely on software.

For proactive host-based security using software, numerous works have argued in favor of system call (syscall) monitoring to detect anomalies during runtime [25], [49], [29], [53], [14], [56]. Researchers have since extended vanilla syscall monitoring to the virtualized case, monitoring in a VM-oblivious fashion [35], [31], [40], [18]. However, syscall monitoring generates a lot of space (syscall logs) and runtime overhead (syscall window matching) and its ability to scale needs to be investigated further for the data center context. Additionally, the approaches discussed in the literature make little attempt to leverage the features of clouds (elasticity, homogeneity/heterogeneity of workloads etc.) to strengthen their designs. Another common approach to inspect VMs from the hypervisor is Virtual Machine Introspection (VMI) [42], [34], [20], [54]. However, VMI also suffers from the same scalability issues mentioned previously [31]. Furthermore, it often requires OS modification [18] or user participation [13] forcing vendors to resort to VMI retroactively after an abuse is reported.

To address these host-level security concerns, we propose an anomaly detection fabric that takes the diverse nature of data center workloads into account and offers customized protection to each client via a paired mechanism of *whitelisting plus blacklisting*. We apply the syscall monitoring work done by earlier researchers to develop a highly scalable cloud-centric system. By monitoring the sequence of system calls originating from a "community" of VMs or containers (set of instances performing the same task), during the training phase, we can develop a profile for normal behavior and

flag any sequence that appears anomalous during runtime (whitelisting). Such flagged sequences are then passed on for further analysis to determine if they are malicious or not. If deemed dangerous, they are prevented from execution on all instances of the community (blacklisting).

Syscall monitoring has been shown to be a promising behavioral malware detection technique, excelling at detecting zero-day threats during their early stages [15] and mitigating DoS attacks. Furthermore, syscall monitoring can also flag malware that can evade traditional defenses [15], such as metamorphic and polymorphic viruses. As a result, our syscall-based solution is accurate with very low false positives and can scale to thousands of servers due to two main design features. Firstly, we compress the data stream (logs of system calls) at its source by the novel use of Cuckoo Filters [21] removing the space overhead commonly associated with syscall solutions. Furthermore, this design choice also has the advantage of reducing the compute overhead of matching against raw syscall logs as anomaly detection is reduced to a mere membership test on the Cuckoo Filter. Secondly, we perform malware analysis on the flagged anomalous sequences on separate instances, which are part of a large distributed fabric owned by the cloud vendor. This allows us to leverage the strengths of clouds and scale the system as needed during high usage times. This also removes any single point of failure from the system. We also discuss ways of extending our design to make detection and mitigation more robust by using syscall monitoring in conjunction with architectural and micro-architectural execution patterns, such as Hardware Performance Counter (HPC) monitoring, which is another promising area of research [19], [32], [48], [24].

Our design is based on two key insights. Firstly, for a particular tenant deployment, groups of VMs are assigned the same set of specialized tasks, which they perform repeatedly. Common examples of these include sets of VMs responsible for graph processing in social networking deployments, MapReduce deployments, database queries in large big data deployments, image processing in medical science deployments etc. Such operations are scaled across a deployment of several "sister" VMs, which we call a *community* [36], all of which exhibit the same signature under normal circumstances. A compromised VM or container, on the other hand, would generate a different signature from its sisters and can be flagged in real-time. The second insight we leverage is that if a small percentage of VMs/containers belonging to a tenant are suspended and their tasks migrated to new instances, the tenant will not notice a major performance degradation caused by the jitter. Hence, borrowing from the concept of "sacrificing the few for the good of the many", we leverage the first few VMs that are the initial targets of a zero-day attack (we cannot protect these VMs in any event) to protect the entire deployment by sharing information on the behavior of the attack and generating a "vaccine" that is broadcast to the entire community. The attacked VMs are then suspended (and can be quarantined for detailed forensic analysis) and substituted with immunized ones that are resilient against the ongoing attack (as they have the vaccine). This prevents a particular malicious behavior from executing on the entire set of sister VMs and contains the infection from spreading across the remaining deployment while maintaining a steady supply of VMs.

Contributions:

- We present the design of a scalable and practical anomaly detection fabric with negligible space and runtime overheads and highly accurate detection rates.
- We introduce the concept of VM communities, which helps with creating strong signatures of normal behavior and collaboration at a global level to detect and thwart attacks.
- We present a model for generating a per community vaccine that can thwart ongoing attacks instantly without disrupting service to the tenant.
- We present a space-efficient way of representing and processing syscalls using Cuckoo Filters.
- Finally, we augment our system with a hardware-based behavioral monitoring scheme.

The rest of the paper is organized as follows; We presents the overall architecture of our syscall monitoring framework in Section II and the evaluations in Section III. We discuss a few possible limitations of our system in Section IV followed by the related works in Section V. Finally, we conclude this work in Section VI.

II. Architecture

Our scheme uses syscall monitoring techniques outlined in past work [35], [40], [33], [37], [39], [25], [49] to develop a comprehensive real-time anomaly/malware detection scheme consistent with the requirements of modern day commercial clouds. Our design is comprised of 3 components namely the Instrumentation Agent, the Detection Agent and the Analysis Agent as shown in Figure 1. The Instrumentation Agent and Detection Agent reside on each host where as the Analysis Agent is distributed across the cloud (in a scalable fabric) to cut down round trip times and save bandwidth. Each Instrumentation Agent monitors and logs the syscalls of each VM and the Detection Agent checks all VMs from the same community for anomalous sequences. The Analysis Agent determines if the anomalies are malicious or not by performing a correlation analysis on anomalous sequences observed across different VMs. Our system is built on the STIDE model [23] of syscall monitoring, which deals with time ordered sliding sequences of syscalls, known as a tuple or a window. This model allows us to perform set operations on the tuples of syscalls, such as union, intersection, addition etc. which will be used during various stages of our design below.

A. VM/Container Communities

The advent of cloud computing has substantially altered conventional programming paradigms. New and emerging distributed and parallel computing schemes, such as MapReduce, Scala and Cuda, have enabled users to unlock the true potential of data centers. However, these new programming models require that a computation be split up and divided across

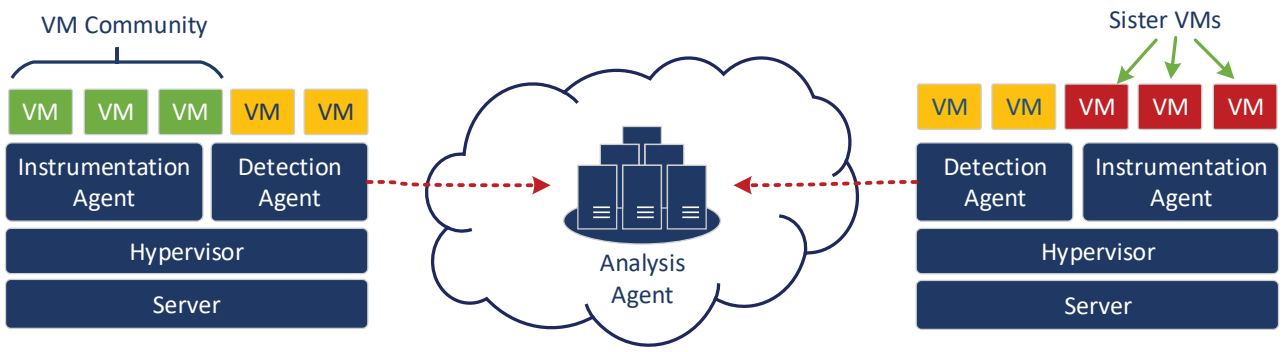

Fig. 1. Bird's eye view of our system call-based design showing various components that comprise the anomaly-detection fabric.

numerous parallel workers specializing in repeatedly performing the same set of tasks. Additionally, it is considered good deployment practice to have specialized VMs/containers performing individual tasks rather than a mix of workloads. This task specialization enables clouds to scale more effectively as only the overloaded instances need to be scaled out based on the increased demand of individual tasks. Modularization of computation, thus leads to "pools" of VMs or containers all performing the same task and exhibiting similar behavior. We call such a pool a VM or container community. A single tenant deployment often comprises multiple communities, such as VMs responsible for the frontend would be considered one community and instances responsible for the backend would be another community. Additionally, one community may be spaced out on multiple different servers or might have all its members on the same cloud node based on the orchestration algorithm of the cloud. Members of the same community are called sister VMs/containers. We assume that identification of VM communities is done via the collaboration of the user and the vendor for simplicity. This would also help the vendor in scheduling the VMs more effectively over the servers by a coming up with a distribution whereby a compute intensive VM is placed next to a VM that has a low demand for processor usage to oversubscribe nodes as much as possible and maximize profits. Community identification could also be done without customer involvement by looking at the VM footprints when the VMs first boot.

B. Building a Community-Wide Profile

System call monitors have to first build a profile of normal behavior for the entity being monitored. This happens during the training phase, whereby all sequences of system calls of a fixed length, are monitored and logged as a set representing normal process behavior. During actual execution, sequences are again monitored and checked against this set for legitimacy. If a sequence falls outside of this set, it is flagged as anomalous. In the case of clouds, before we begin the training phase we first divide a tenant deployment into communities, such that sets of VMs are categorized as being responsible for certain specialized tasks. Once these communities have been identified, we start profiling each VM in a community separately. We initiate the profiling process by building profiles for each sister VM individually and hence we can cover more legitimate sequences of syscalls by merging all profiles when the training phase ends (union of all observed syscall tuples). This tactic, which leverages the elasticity of clouds, allows us to work with shorter training periods by increasing the number of sister VMs being profiled during the training phase. The length of the training phase depends on the vendor. Typically, longer training intervals yield more accurate detection rates and since we parallelize the training we can exhibit high accuracy despite shorter intervals should the vendor choose so. Furthermore, the vendor can and should choose different tuple sizes for each community to customize the features of the monitoring framework based on the nature of the workload. This design choice is consistent with past works, which have shown different window sizes to be optimal for different types of workloads, such as 6, 10, 11 etc. [12], [25], [29]. The profile building task is the responsibility of the Instrumentation Agent.

C. Compressing the Data Stream

The first challenge that we address in our design is that of space overhead generated due to syscall monitoring. As shown in Section III, the size of raw syscall logs becomes so huge after just a few minutes into the production phase that it makes the approach impractical and averse to scaling. Hence, we compress the data stream i.e., the sequences of syscalls by using a Cuckoo Filter. A Cuckoo Filter (an improved version of Bloom Filters with faster lookup times) is a simple data structure that can house large amounts of data by using its own space-efficient intermediate representation (array of bits) based on hashing and respond to queries about set membership pertaining to the data it has housed (has no false negatives). The Cuckoo Filter takes a string as input and maps its hash onto an array by setting numerous disjoint indexes to 1. This scheme allows us to represent the complete legitimate behavior of a tenant in the form of a single array and search for anomalies by performing hash operations instead of parsing an entire log. Each time a new sequence is observed, it is checked against the Cuckoo Filter and if absent from the profile, is flagged as anomalous. Additionally, the Detection Agent,

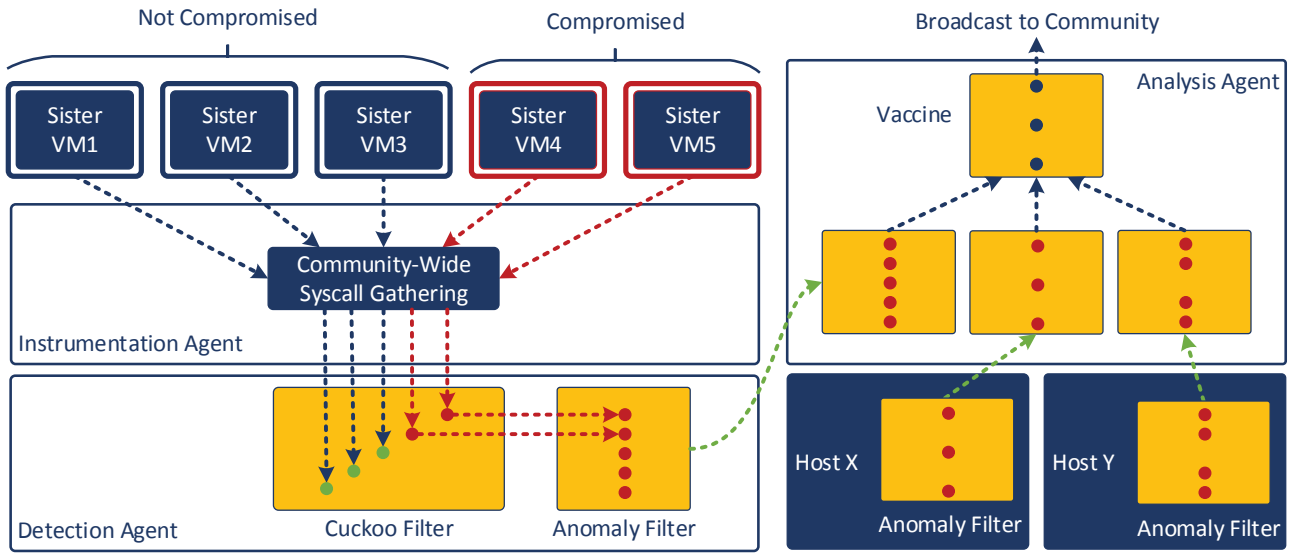

Fig. 2. Detailed schematic diagram showing how anomalies are flagged individually but examined collectively to determine if they are malicious.

which is tasked with the detection of anomalous sequences, further minimizes the space overhead by using just a single Cuckoo Filter per community for all member VMs running on that host to check for compliance against normal behavior.

D. Defending the Community

Researchers have argued that sharing information about process infections amongst collaborating nodes can provide security against said infections [36], [39]. Cloud computing is perfect for such a collaborative community-based protection model, as jobs are parallelized and distributed onto a large number of worker nodes all performing the same operation, MapReduce and graph processing are excellent examples of this. Furthermore, vendors strongly advise against using mixed workloads on top of single instances [12], which is also in line with the community-based paradigm. Hence, porting the community model to cloud implies that the first few instances that get attacked share the characteristics and features of the attack with the remaining sister VMs from the community. In our design, these features of an ongoing attack are characterized by time ordered sequences of syscalls. If the same anomalous sequence of syscalls is observed across a predetermined threshold number of sister VMs, it can be flagged as malicious and broadcast to the entire community. Of course, this anomalous sequence could be a legitimate sequence that simply got missed during the training phase, however, the probability that a missed legitimate sequence starts appearing on a threshold number of VMs all at the same time is small. Determining which sequences are anomalous is the responsibility of the Detection Agent.

E. Generating a Vaccine

As mentioned, the vaccine alerts other VMs to dubious or malicious sequences of syscalls. When a Detection Agent comes across a sequence that is not present in its Cuckoo Filter, it marks the sequence and saves it in another Cuckoo Filter, known as the Anomaly Filter. The Detection Agent inserts all anomalous tuples into the Anomaly Filter and after a fixed time interval elapses (known as the monitoring cycle), which is a tunable metric representing how tolerant the system is to attacks, the Detection Agent sends its Anomaly Filter to the Analysis Agent for further inspection. The Analysis Agent is deployed as a distributed cloud tenant in our design, which allows it to scale under load and have a divided footprint across the data center to prevent any single point of failure. The Analysis Agent receives one Anomaly Filter per VM, at the end of a monitoring cycle, from each Detection Agent supervising a community giving the Analysis Agent a global view of the state of the community. Since the Anomaly Filter is actually a Cuckoo Filter, its simply an array of bits, the Analysis Agent can perform some basic counting operations on it. Specifically, the Analysis Agent adds together all Anomaly Filters it has received from a community, which can be understood as the union of all anomalous tuples, and determines which indexes cross a certain predetermined threshold value (called vaccine threshold). It then creates another Cuckoo Filter and sets the corresponding indexes, which have crossed the threshold value, to 1 and marks all others as 0. The resulting filter is what we call a vaccine and houses the anomalous sequences, which have already appeared on a threshold number of VMs in the community. The vaccine thus generated can be thought of as a blacklist and the Cuckoo Filter for normal behavior is the whitelist. The entire operation is depicted in Figure 2. Upon vaccine generation, the Analysis Agent broadcasts it to all Detection Agents in the community and requests the cloud orchestration framework to instantiate substitute VMs in place of the compromised ones. As we show in Section III-G, the vaccine threshold can be tweaked to reduce the detection delay at the cost of false positives.

F. Containing the Infection

Once a Detection Agent receives the vaccine, it checks all running sequences in each sister VM against the vaccine to make sure that it too has not been compromised. If no hits are observed the VM is believed to be contamination-free and allowed to continue execution otherwise it is reported to the Analysis Agent, which in turn notifies the orchestration framework of the VM's compromise. As a result, the spread is contained before it causes further devastation. The client experiences little or no degradation as substitute VMs are quickly spun up to meet the demand.

G. Extending the Base Design

We also propose some ways to extend the base design of our system to make it more robust and efficient.

1) Secondary Detection via HPCs:: Recent works [19], [32], [48], [24] have shown that architectural and micro-architectural execution patterns can be used to profile application behavior. One prominent hardware component in this category is the set of Hardware Performance Counters (HPCs), which can be used effectively to detect a particular behavior on a host. HPCs are low-level registers that keep track of hardware level and even system level events during a program's execution. Common examples include number of cache misses, number of TLB flushes etc. We propose to use these HPCs in combination with our syscall-based design. HPCs can be monitored in a scalable and speedy way with very little overhead, giving them the same properties as those of our syscall monitoring fabric. They do not violate clients' privacy and do not interfere with normal program execution. As we show in Section III, it is easy to set up HPC monitoring with syscall monitoring. Hence, HPCs can serve as an excellent detection metric to "beef up" the security of the system. Targeted attacks on individual VMs that do generate an anomalous sequence but don't get flagged in the thresholding process can be easily detected via HPCs. A simple classifier can be trained to identify anomalies based on observed HPC values during monitoring cycles as we show in Section III-I and III-J. The classifier can be used sparingly under special circumstances in order to avoid load on the system.

2) Updating the Behavioral Profile:: The Detection Agent sends all anomalous sequences to the Analysis Agent however, not all sequences are malicious and some are in fact legitimate sequences that got missed during the training phase. For instance, a very rare control flow path in the application could result in the occurrence of such a sequence. Our system keeps track of all anomalous sequences observed by a community that did not pass the threshold for maliciousness, as mentioned above, and were only observed on very few sister VMs (a benefit of the global visibility of the Analysis Agent). Even though the sequence could still be malicious, for instance from a stealthy malware, we would like to investigate it further and add it to the normal profile of the community if it turns out to be a benign sequence. This can be done by checking the said sequence against a database of known blacklisted syscall sequences [16] or by building a model of the community's syscall behavior and automatically generating "bad" sequences, as done by Giffin et al. [26], and comparing against those or another venue is to use our secondary HPC-based detection system. If the syscall sequence comes out clean we propose to add it to the community's base profile after a certain time interval has elapsed and the sequence is recurring.

III. Evaluation

As mentioned previously, there have been many proposals in recent literature as to the mechanism for syscall monitoring. However, to keep things simple, we resort to the Linux Perf tool, which is much faster than strace [6] and allows us to work directly with the KVM Hypervisor without any modifications to the base hypervisor. Furthermore, Perf also allows us to monitor Hardware Performance Counters, which we use as a secondary detection mechanism. Our goal is not to compare the various syscall approaches, in fact any of the described approaches can be used in our system.

For the experiments we chose an Intel i7 3.40GHz x8 machine with 8GB of RAM, 8.5GB of swap space and a 500GB hard drive. For the host, we had Ubuntu 15.10 based on 4.2.0 kernel version. For guests, we primarily stuck with Cloudera Quickstart VM 5.5 which uses CentOS 6.7 however, other OSes work equally well. The version of Perf was 4.2.8-ckt5. Where applicable, we chose the window size to be 10, monitoring cycle to be 10s and vaccine threshold to be 4 percent.

A. Runtime Overhead of Instrumentation

One primary concern of our design was the runtime overhead generated by the instrumentation process. Since our solution is designed for clouds where customers expect low job completion times, we wanted to make sure our system did not slow down jobs to unacceptable levels. To this end, we chose 5 representative workloads, namely Hadoop MapReduce Image Processing (HIPI), Kernel Compile, Video Encoding/Decoding, File Copying and Encryption/Decryption, and ran them with and without system call instrumentation in place. During this exercise we observed the job completion times of each task to get a sense of the runtime overhead added by the instrumentation process. Figure 3 shows the average numbers for the performed experiments over numerous runs. As demonstrated, the added overhead in the job completion times of each of the 5 cloud workloads is negligible. This means that the instrumentation phase of our system is extremely low-overhead and resultantly scale-friendly. Furthermore, in general the overhead of Linux Containers (LXC) was found to be lower than that of VMs.

B. Runtime Overhead of Detection

Our design is such that detection merely involves a constant time operation on a Cuckoo Filter, which is a very high performance data structure. However, to ascertain the scalability of our system, we wanted to ensure that the detection phase, like the instrumentation phase, incurs tolerable overheads and

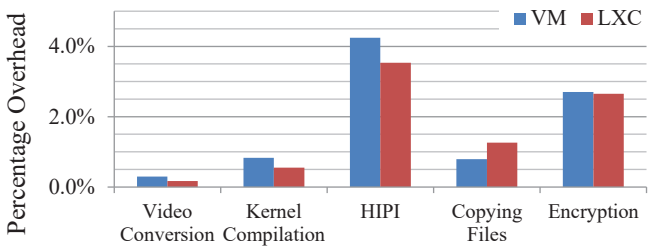

Fig. 3. Runtime overhead of the system call instrumentation process over different workloads.

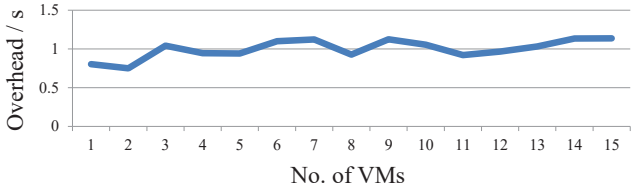

Fig. 4. Runtime overhead of the detection phase over a workload generating around 2 million system calls.

maintains this constant time characteristic when scaled to higher VM counts per node. Hence, we ran the detection mechanism across a large number of processes running simultaneously, as shown in Figure 4. These delays were incurred by the detection process, which checks a tuple of system calls for membership against the Cuckoo Filter and adds any anomalies to the Anomaly Filter, for roughly 2 million system calls. Results show that even when the VM count on a single node is high, the system maintains its characteristic of constant time detection backing our claims of scalability. Hence, in terms per node compute overhead, including both instrumentation and detection, our system is highly economical making it a perfect candidate for host-based security in clouds as it has no slowdown effect on the VMs.

C. Space Overhead of Instrumentation/Detection

One prominent feature of our approach is the space efficient design, which is achieved via the use of Cuckoo Filters. The idea behind Cuckoo Filters is to absorb a large dataset and store it in a space efficient intermediate representation. Despite the fact that the actual dataset is lost when converted to a Cuckoo Filter, we can still run constant time queries pertaining to the dataset such as the presence and absence of certain elements in the dataset. For our purposes, the Cuckoo Filter houses the entire normal behavior of the community in question and we can run queries against it to find out if a sequence is present or missing. This approach offers a tremendous benefit in terms of space usage compared to raw logs as done traditionally. Figure 5 attempts to quantify the benefits of using a Cuckoo Filter as opposed to raw syscall logs. The size of a Cuckoo Filter remains pretty much constant even when the system has been running for a decent amount of time as opposed to raw logs, which start to incur a large amount of space overhead. This result, together with the runtime overhead presented above, affirms our claims to

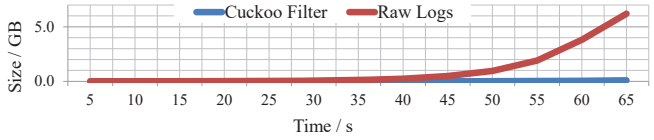

Fig. 5. Comparison of the space overhead of our Cuckoo Filter-based approach as opposed to using raw logs of system calls.

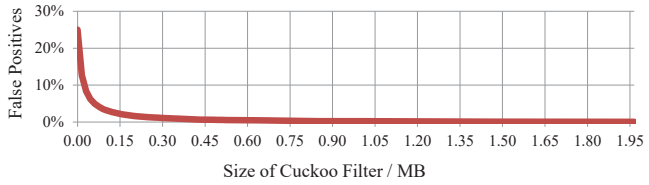

Fig. 6. Relationship between the size of the Cuckoo Filter and false positives.

the overall scalability of our system.

D. Size of Cuckoo Filter vs Number of False Positives

As with other similar hash-based data structures, Cuckoo Filters offer a tradeoff between size of the filter and the degree of false positives. With larger sizes, Cuckoo Filters generate smaller number of false positives as they are able capture and record a wide range of legitimate behavior while minimizing overlapping (two strings hashing to the same indexes). Smaller filters on the other hand perform poorly as they are poor representatives of the entire spectrum of normal behavior. However, for our purposes, we wanted to investigate what size of the Cuckoo Filter generates acceptable false positives so that the network overhead can be minimized. If the size at which false positives become acceptable is impractical then Cuckoo Filters must be replaced with other alternate data structures, such as Bloom Filters, which have been used by other researchers to reduce the space overhead of various systems [25]. In Figure 6, we show that our choice of a Cuckoo Filter is indeed justified as false positives can be reduced to negligible levels by using very nominal sizes of the filter. With very small filters (<0.1MB) the corresponding false positives are large due to the reason mentioned above. With a 1MB Cuckoo Filter, our system exhibits almost 0% false positives, which is remarkable given that we are able to represent the entire normal behavior of an application in 1MB as opposed to tens of GB of raw syscall logs needed otherwise. Also, worth mentioning is that on a single node, we only have one such Cuckoo Filter per community, which means that the per node footprint of our system is very small.

E. Length of Training Interval vs Number of False Positives

Another important metric for evaluation is the length of the training interval and how it impacts the accuracy of detection. Since we are in a data center setting, the vendor will usually not have enough time as the customer would want to move onto the normal phase very quickly. Hence, to measure this relationship between training interval and accuracy, we plot the training interval duration on the X-axis and the corresponding

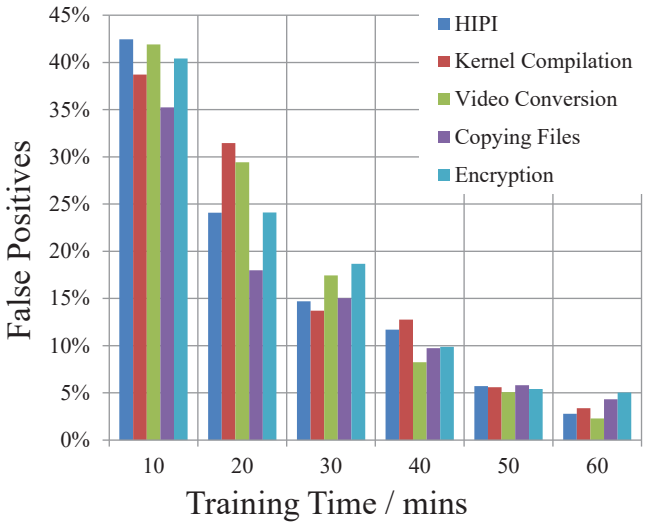

Fig. 7. Relationship between the length of the training interval and false positives.

percentage of false positives on the Y-axis in Figure 7. As can be seen, there is decreasing trend in the number of false positives as the training window increases. The percentage can be brought down to acceptable levels within the hour for various workloads. If numerous sister VMs are being simultaneously profiled then this interval can be reduced even further. Additionally, if workloads are similar across tenants, the vendor can leverage the profile of one tenant and use it directly (or with minimal training) for the other one.

F. Length of Window/Tuple of System Calls

Size of the tuple/window of system calls has been the subject of contention in many prior works. Various researchers have used different window sizes by empirically justifying their use of the size in the form of low false positives. Size 6, 10, 11 and even 20 has been discussed and both pros and cons of each size have been presented. Through our experiments as shown in Figure 8, we found that as the window/tuple size increases, it loses the granularity at which system call patterns change. Thus it is unable to detect those patterns and flag suspicious behaviour. This causes an increase in false negatives. Hence, a balance needs to be struck between speed, size and accuracy, which is different for different types of workloads. Given this consideration we make this metric a tunable feature of our design, with the vendor having the flexibility to work with larger or smaller sizes based on the nature of the jobs being run inside the community and training time available to the vendor. It is pertinent to mention here, that for our case, with larger window sizes, the system works just as fast as the smaller window sizes because of our use of the Cuckoo Filter. This is because the filter stores the sequences of syscall in its own intermediate representation, which is pretty much the same for any size of the tuple, and does away with the concept of an exhaustive entry by entry perfect matching approach. Also, the numbers shown in Figure 8 improve substantially if a larger training window

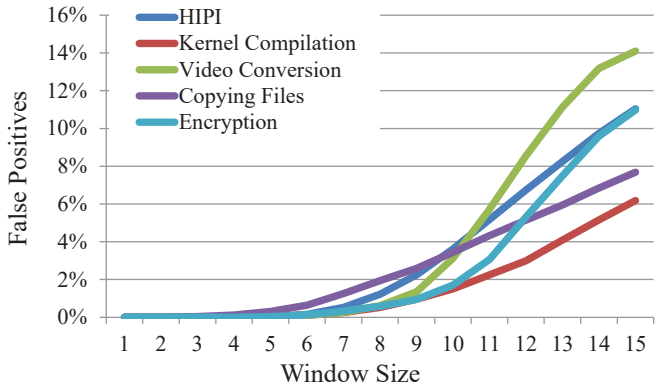

Fig. 8. Varying rate of false negatives for different window/tuple sizes for different workloads.

is used, however, since our system is designed for clouds and customers want to move to the production environment quickly, we limited ourselves to a training window of 1 hour to get these results.

G. Vaccine Threshold and Monitoring Cycle vs False Positives, Detection Rate and Malware Spread

The threshold value used to generate the vaccine is again a tunable parameter. Smaller values result in quicker generation of the vaccine however, this improved latency comes at the cost of increased false positives. In contrast, a larger threshold value would mean that the Analysis Agent will wait to see more anomalous sequences in each of the incoming Anomaly Filters before flagging them as malicious. This would result in a slight delay that might allow the malware to spread across more members of the community. However, this will also reduce the chance of false positives as a rare legitimate sequence, which was missed out during the training phase, will not get flagged. Fig 9 captures this trade-off inherent to our system. The bars on the top are associated with the Y-axis on the right and the bars below are tied to the Y-axis on the left. Along similar lines, we ran a variant of this experiment to test the relationship between the spread of malware to that of vaccine threshold and monitoring cycle. To explore this relationship we exploited a vulnerability in the guest OS, which allowed us to execute the contents of a UDP packet. Once the payload would execute, it would connect to a CnC server and search for jpg files in the gest OS file system and send their paths to the CnC server. The malware would then infect other VMs by sending the same UDP packet to other VMs with similar IP prefixes in the subnet. We repeated this scenario numerous times with three different attack rates (5, 10 and 20 second wait times before sending the UDP packet to another VM mimicking a slow, normal and fast malware respectively). We deployed a total of 15 VMs and tested for three different vaccine thresholds (10, 40 and 70% of the total VMs) while keeping the value of the anomaly window constant at either 5, 10 or 15 seconds for the duration of the experiment. As shown in Fig 10, smaller thresholds and shorter monitoring cycles can thwart an attack very quickly preventing a large part

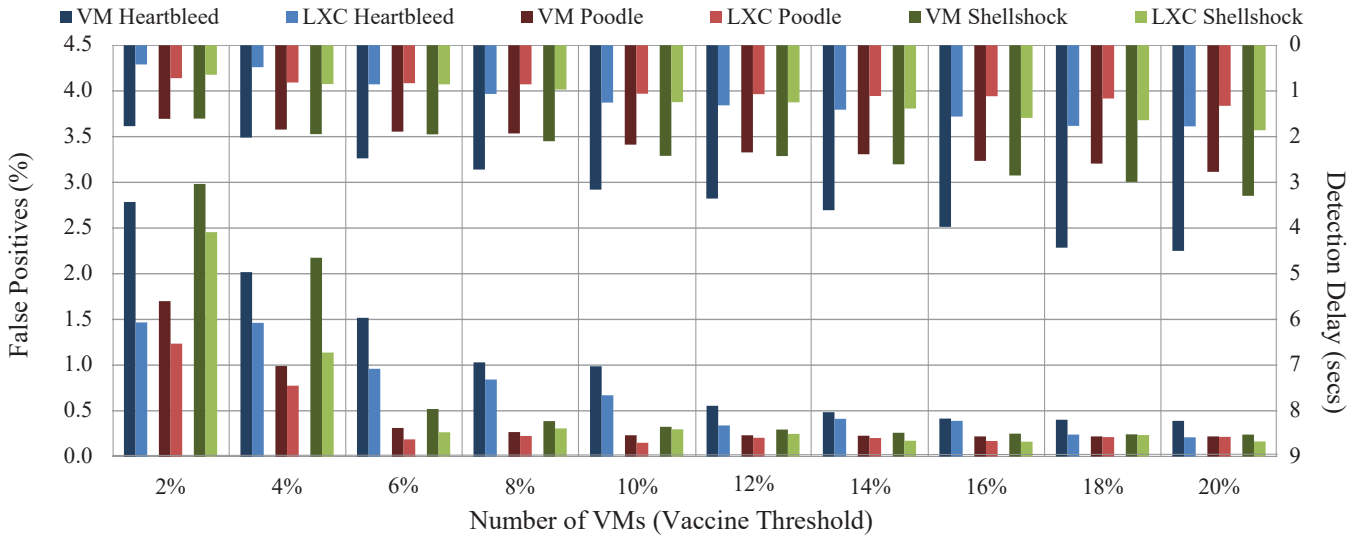

Fig. 9. Change in detection delay and false positives as the vaccine threshold is changed. Y-axis on the right side is for the bars drawn from top to bottom.

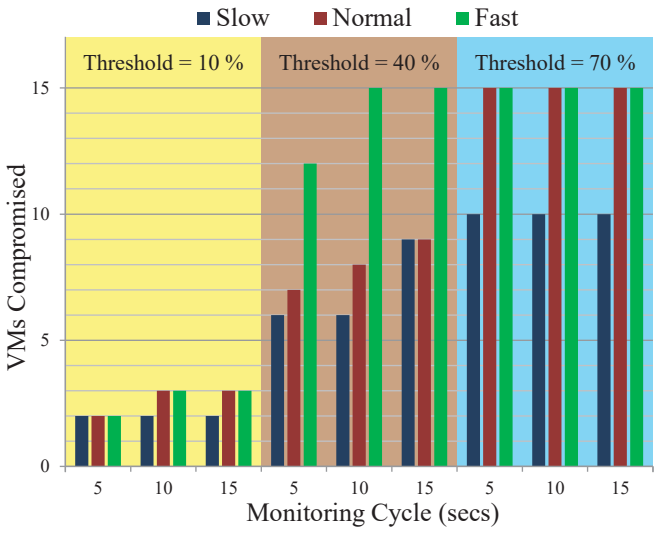

Fig. 10. Vaccine threshold and monitoring window effect the rate at which the vaccine is generated, which in turn prevents malware from spreading to other VMs/containers.

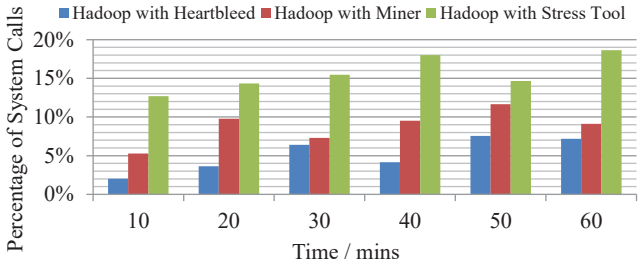

Fig. 11. Demonstrating the additional syscalls generated by infected versions of a Hadoop workload over 60 minutes of runtime.

of the deployment from being infected. Larger thresholds and longer monitoring cycles, despite giving low false positives as shown previously, take longer to detect the malware and allow it to infect more VMs in the community.

H. Anomaly Detection of Infected Workloads

To test the entire system as a whole and check the efficiency of malware detection, we ran four versions of the MapReduce Image Processing workload side by side. For the first case, we used the unmodified version of the workload out of the box however, for the remaining three cases we infected the workload with a particular type of malware, such as a Heartbleed vulnerability scanner (shown in green), a Bitcoin mining trojan (shown in red) and finally a small stress tool (shown in blue), which attempts to waste system resources needlessly. We then profiled each of the four workloads for a total of 60 minutes observing the number of sequences generated over 10 minute intervals by each workload individually. As shown in Figure 11, the three compromised versions of the workload generated substantially more sequences as opposed to the normal version, with the stress tool generating the highest number of additional tuples followed by the Bitcoin miner and the Heartbleed scanner coming in at last. These additional sequences are successfully flagged by the anomaly detection mechanism and passed onwards for malware analysis.

I. Overhead of HPC Monitoring Extension

As an extension to our main design, we propose adding a secondary anomaly detection system based on Hardware Performance Counters. To demonstrate the feasibility of the HPC plus syscall approach working in combination, we ran both systems together and noted the job completion times for various cloud-representative workloads. Since the Perf tool allows us to monitor HPCs as well, this turned out to be a straightforward extension with numerous benefits. As shown in Figure 12, the added instrumentation overhead, resulting from HPC monitoring, is affordable for most applications and hence, the HPC-based detection can be used in a supportive role with our main syscall-based approach.

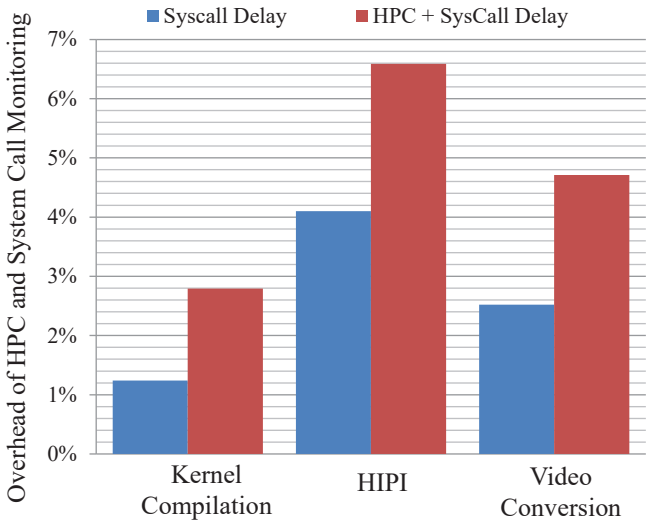

Fig. 12. Overhead of Hardware Performance Counter monitoring in combination with syscall monitoring as opposed to only syscall monitoring.

J. Effectiveness of HPC Extension

To test the effectiveness of our HPC-based extension, we trained a classifier to detect a mining trojan on a running VM that does not spread in order to evade the thresholding process. A total of 26 HPCs (all supported by our Intel machine) were used during this experiment, most of which targeted cache and TLB operations and some of which were generic such as branch misses, context switches etc. Since we did not know the underlying distribution of the various performance counters altered during mining, we tried different non-parametric classifiers including k-Nearest Neighbor (k-NN), Bagged and Multiclass Decision Trees etc. We found that ensemble-based classifiers outperformed others at least as far as mining trojans are concerned. During the training phase, we trained the classifier over a wide range of data center workloads borrowed from CloudSuite v3.0 [22], SPEC 2006 benchmark [7] and common Hadoop tasks with and without the miners running in the background. In the test phase, we fed the classifier unseen mining trojans. The classifier gave an average F-score of 96.37% with a false positive rate of 6.24% and a false negative rate of 1.25%. For an open world setting (flagging unseen trojans), these are decent results. Furthermore, to match a new sample the classifier took less than 10ms on average. Figure 13 shows HPC-based signatures for various cloud workloads as well as the miner. The two subgraphs show a live-trace of a particular counter's value during the execution of a CPU-based miner and four non-mining applications; namely data caching (memcached server), AI (game of Go), H264 (hardware video encoding) and NAMD (molecular dynamics). The values of other counters and workloads have been omitted for clarity. Signatures were generated by running the workloads in a VM and profiling the HPCs via the Perf tool. As can be seen, the signatures are vividly discernible, which means that HPCs generate a strong behavioral profile that can be used to flag malware in real time.

IV. Attacks and Limitations

One primary concern for previous syscall-based systems has been the mimicry attack [26],[52], where by an attacker tries to carry out malicious activities while mimicking legitimate sequences of syscalls. However, since our syscall mechanism treats a VM as a blackbox and not as a group of separable processes running on an OS, it is much more robust against mimicry attacks, as demonstrated by other researchers as well [12]. Additionally, mimicry attacks perform poorly against systems which incorporate the arguments of the syscalls into detection scheme [12]. Given the flexibility of our design and Perf's ability to collect arguments of syscalls, we implemented this feature into our design as a simple and straightforward extension making it more robust. Still, it is possible, albeit highly improbable and very hard for an attacker to launch a mimicry attack against our system so we list it as a limitation here.

Targeted attacks on a single instance (to steal data for instance) will not be detected by our syscall system, since our focus is on protecting a larger deployment from spreading attacks such as worms, as opposed to individual VMs. However, the HPC-based extended framework can detect these and other single host attacks as shown in Section III-J. Similarly, VMs running heterogeneous workloads, in which case the notion of a community will not hold, are not applicable to our system. Though considered against best practices, such deployments can be found in clouds and our system will not work on such instances.

Another possibility is for an attacker to subvert the kernel running inside the OS and modify the tracing infrastructure by changing the location and layout of critical kernel data structures, syscall service routines, function pointers etc., which are needed by various monitoring tools that gather system call dumps. However, the attacker would first need to get root access, which will generate an anomalous sequence and get flagged. However, it is still possible for an attacker, though quite hard, to achieve this via a mimicry attack or by avoiding syscalls altogether making it a limitation of our system.

Finally, it is pertinent to point out that our system should always be deployed in combination with network-based defenses, as it is difficult for a host-based anomaly detection scheme to detect attacks originating from and targeting networks.

V. Related Works

The literature is abundant with research focusing on system call monitoring in the traditional case [25], [49], [29], [53], [14], [56], [15], [16], [26] and the virtualized case [33], [35], [40], [12], however, no work prior to ours, has targeted a fully scalable and extensible, cloud centric system call monitoring framework that can be practically deployed in a commercial data center. The closest work to ours is from Alarifi and Wolthusen [12], who present a syscall analysis

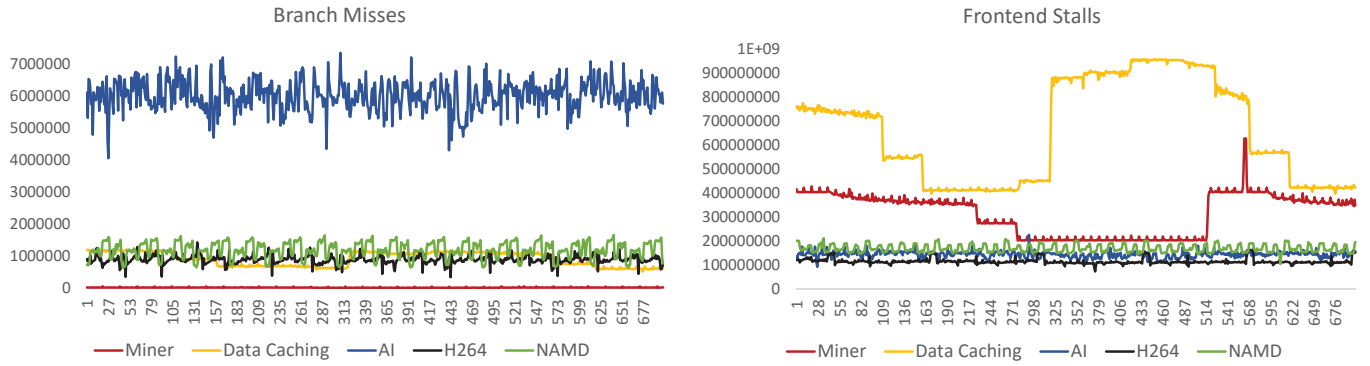

Fig. 13. Behavioral profile of a Litecoin CPU miner and four representative cloud-based CPU applications. The x-axis shows time in increments of 100 milliseconds. The miner is shown in red.

framework for the IaaS model based on strace [10]. Their design, however, does not target scalability and does not leverage features innate to the cloud like ours. Furthermore, they do not address the problem of space overhead resulting from the syscall instrumentation as we do. Additionally, we also present avenues for the extension of our work, such as the added HPC monitoring framework, which itself has been the subject of various research endeavors [19], [32], [48], [24], however it has not been discussed in the context of syscall monitoring prior to this work. Others have demonstrated how to sample and collect system calls in virtual environments in a VM-oblivious fashion [35], [33], [40] and serve as inspiration to this work.

The fundamentals of the syscall monitoring were laid out by Forest et al. [23], Hofmeyr et al. [29] and Warrender et al. [53], in their pioneering works by demonstrating how sequences of system call originating from a process could be monitored for anomalous behavior. The systems involved a training phase during which sequences of system calls were recorded forming a set, which served as the baseline for "normal" behavior. Once actual execution began, outside of the training phase, all sequences that were absent from the set of normal behavior were flagged as anomalous. However, the approaches discussed had an inherent tradeoff pertaining to the duration of the training period; larger sizes resulted in lesser false positives but higher false negatives where as smaller durations would substantially increase false positives. Subsequently, Locasto et al. [36] demonstrated a way to reduce false positives without compromising on the false negatives by leveraging the uniformity of behavior in "application communities". Other researchers [39], [37] explored the temporal correlation between flagged system calls across distinct instances of the same program to improve detection rates further.

The second theme, found in our work, is that of cloud abuse. Researchers have tried to identify the vulnerabilities in clouds and how they can be abused to attack others [51], [30], [46], [57]. This has led to numerous works showing how data centers can be exploited in novel ways by building file sharing applications [50], unlimited storage banks [38], email-based storage overlays[45] and mining botnets [4] on top of regular cloud services in a manner oblivious to the cloud vendor. Similarly, various research endeavors have tried to identify the existence of covert and side channels between VMs that are coresident on the same server [28], [27], [43], [55] or share underlying network infrastructure [47], which also create opportunities for cloud abuse and can be used to great effect for data exfiltration.

To detect this cloud abuse at the host-level, other researchers have proposed a technique known as Virtual Machine Introspection (VMI), which gives vendors the ability to look inside the contents of a VM and detect unwarranted activity [42], [34], [20], [13], [31], [54]. However, VMI sometimes requires user participation [13] or OS modification [41] and can have large overheads[31], [20]. Additionally, traditional VMI approaches lack the scalability needed in the cloud setting, which makes them unsuitable for our case. Resultantly, cloud vendors mostly resort to VMI retroactively once an abuse is reported to have occurred. Others have tried to bridge the semantic gap between the guest's view of its internal state and the hypervisor's view of the VM [31]. Finally, a separate body of literature aims at creating secure execution environments, such as SecVisor [44] and Secure Virtual Architecture [17], however these approaches require modifications to the OS or the application and can be used in combination with our approach.

VI. Conclusion

Given the rise in instances of cloud abuse, robust and scalable host-based anomaly detection systems are needed that can scale to cover the overarching expanse of clouds and provide customized protection to each client. In this paper, we present one such scheme, which has a cloud-centric design – in that it specifically targets clouds and also leverages the properties of clouds, the scheme is highly scalable – it has a negligible per node space and runtime overhead, the scheme is extensible – such as adding more features and properties for enhanced detection, and the scheme is customizable – in that the vendor can tune various properties of the system to adjust detection in accordance with the nature of the client workloads. Our system makes use of common profiling tools

easily available in off the shelf hypervisors. The proposed syscall monitoring-based approach uses Cuckoo Filters to capture normal behavior and generates malware signatures, based on the temporal correlation between instances of anomalous behavior across machines, which protect tenant deployments from attacks. We perform a thorough empirical analysis to test our system and substantiate our claims of scalability, effectiveness and extensibility.

REFERENCES

[1] venturebeat.com. March 2010. https://tinyurl.com/y98ndrm.
[2] deepdotweb.com. August 2014. https://tinyurl.com/grbjuvt.
[3] readwrite.com. April 2014. https://tinyurl.com/goqkxdx.
[4] wired.com. July 2014. https://tinyurl.com/mowzx73.
[5] awsinsider.com. September 2015. https://tinyurl.com/zm3pzjq.
[6] Julia Evans Blog. March 2015. https://tinyurl.com/q8mes75.
[7] spec.org. 11 2015. https://www.spec.org/benchmarks.html.
[8] arbornetworks.com. January 2016. https://tinyurl.com/h9a58ca.
[9] cisco.com. January 2016. https://tinyurl.com/je693kw.
[10] Linux Man Page. January 2016. https://tinyurl.com/jo4a7x7.
[11] secpoint.com. January 2016. https://www.secpoint.com/penetrator.html.
[12] S. S. Alarifi and S. D. Wolthusen. Detecting anomalies in iaas environments through virtual machine host system call analysis. In *ICITST 2012*.
[13] H. W. Baek, A. Srivastava, and J. E. van der Merwe. Cloudvmi: Virtual machine introspection as a cloud service. In *2014 IEEE IC2E*.
[14] J. B. D. Cabrera, L. M. Lewis, and R. K. Mehra. Detection and classification of intrusions and faults using sequences of system calls. *SIGMOD Record*, 30(4).
[15] R. Canzanese, S. Mancoridis, and M. Kam. System call-based detection of malicious processes. In *QRS 2015*.
[16] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *CCS 2002*.
[17] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: a safe execution environment for commodity operating systems. In *SOSP 2007*.
[18] A. W. Dehnert. Using vprobes for intrusion detection.
[19] J. Demme et al. On the feasibility of online malware detection with performance counters. In *ISCA'13*.
[20] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS 2008*.
[21] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *CoNEXT 2014*.
[22] M. Ferdman et al. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *ASPLOS*, 2012.
[23] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *(S&P 1996)*.
[24] A. Garcia-Serrano. Anomaly detection for malware identification using hardware performance counters. *CoRR*.
[25] A. Gehani, B. Baig, S. Mahmood, D. Tariq, and F. Zaffar. Fine-grained tracking of grid infections. In *Grid 2010*.
[26] J. T. Giffin, S. Jha, and B. P. Miller. Automated discovery of mimicry attacks. In *RAID 2006*.
[27] M. M. Godfrey and M. Zulkernine. Preventing cache-based side-channel attacks in a cloud environment. *IEEE Trans. Cloud Computing*, 2(4).
[28] Y. Han, T. Alpcan, J. Chan, C. Leckie, and B. I. P. Rubinstein. A game theoretical approach to defend against co-resident attacks in cloud computing: Preventing co-residence using semi-supervised learning. *IEEE Trans. Information Forensics and Security*, 11(3).
[29] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3).
[30] J. Idziorek and M. Tannian. Exploiting cloud utility models for profit and ruin. In *IEEE CLOUD 2011*.
[31] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *CCS 2007*.
[32] S. Kompalli. Using existing hardware services for malware detection. In *IEEE SPW 2014*.
[33] M. Laureano, C. Maziero, and E. Jamhour. Protecting host-based intrusion detectors through virtual machines. *Comput. Netw.*, 51(5).
[34] T. K. Lengyel, J. Neumann, S. Maresca, B. D. Payne, and A. Kiayias. Virtual machine introspection in a hybrid honeypot architecture. In *CSET '12*.
[35] B. Li, J. Li, T. Wo, C. Hu, and L. Zhong. A vmm-based system call interposition framework for program monitoring. In *ICPADS 2010*.
[36] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software self-healing using collaborative application communities. In *NDSS 2006*.
[37] D. J. Malan and M. D. Smith. Exploiting temporal consistency to reduce false positives in host-based, collaborative detection of worms. In *WORM 2006*.
[38] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. R. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security Symposium, 2011*.
[39] A. J. Oliner, A. V. Kulkarni, and A. Aiken. Community epidemic detection using time-correlated anomalies. In *RAID 2010*.
[40] K. Onoue, Y. Oyama, and A. Yonezawa. Control of system calls from outside of virtual machines. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08.
[41] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *(S&P 2008)*.
[42] B. D. Payne and W. Lee. Secure and flexible monitoring of virtual machines. In *(ACSAC 2007)*.
[43] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS 2009*.
[44] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP 2007*.
[45] J. Srinivasan, W. Wei, X. Ma, and T. Yu. EMFS: email-based personal cloud storage. In *NAS 2011*.
[46] S. Sundararajan, H. Narayanan, V. Pavithran, K. Vorungati, and K. Achuthan. Preventing insider attacks in the cloud. In *ACC 2011*.
[47] R. Tahir et al. Sneak-Peek: High Speed Covert Channels in Data Center Networks. In *IEEE INFOCOM 2016*.
[48] A. Tang, S. Sethumadhavan, and S. J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *RAID 2014*.
[49] D. Tariq et al. Identifying the provenance of correlated anomalies. In *SAC 2011*.
[50] R. G. Tinedo, M. S. Artigas, and P. G. López. Cloud-as-a-gift: Effectively exploiting personal cloud free accounts via REST apis. In *2013 IEEE CLOUD*.
[51] L. M. Vaquero, L. Rodero-Merino, and D. Morán. Locking the sky: a survey on iaas cloud security. *Computing*, 91(1).
[52] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS 2002*.
[53] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *1999 IEEE (S&P 1999)*.
[54] F. Westphal, S. Axelsson, C. Neuhaus, and A. Polze. Vmi-pl: A monitoring language for virtual platforms using virtual machine introspection. *Digital Investigation*, 11, 2014.
[55] Z. Wu, Z. Xu, and H. Wang. Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Trans. Netw.*, 23(2).
[56] X. Xu and T. Xie. A reinforcement learning approach for host-based intrusion detection using sequences of system calls. In *ICIC 2005*.
[57] K. Zunnurhain. FAPA: a model to prevent flooding attacks in clouds. In *Proceedings of the 50th Annual Southeast Regional Conference, 2012*.