PrivateEye: Scalable and Privacy-Preserving Compromise Detection in the Cloud

Behnaz Arzani[1], Selim Ciraci[2], Stefan Saroiu[1], Alec Wolman[1], Jack W. Stokes[1], Geoff Outhred[2], Lechao Diwu[2]

[1]*Microsoft Research,*[2]*Microsoft*

Abstract – Today, it is difficult for operators to detect compromised VMs in their data centers (DCs). Despite their benefits, the compromise detection systems operators offer are mostly unused. Operators are faced with a dilemma: allow VMs to remain unprotected, or mandate all customers use the compromise detection systems they provide. Neither is appealing: unprotected VMs can be used to attack other VMs. Many customers would view a mandate to use these detection systems as unacceptable due to privacy and performance concerns. Data from a production cloud show their compromise detection systems protect less than 5% of VMs.

PrivateEye is a scalable and privacy-preserving solution. It uses summaries of network traffic patterns obtained from the vSwitch, rather than installing binaries in customer VMs, introspection at the hypervisor, or packet captures. It addresses the challenge of protecting all VMs at DC-scale while preserving customer privacy and using low-signal data. We developed PrivateEye to meet the needs of production DCs. Evaluation on VMs of both internal and customer VM's shows it has an *area under the ROC curve* – the graph showing the model's true positive rate vs its false positive rate – of 0.96.

1 Introduction

Data center (DC) VMs today are largely unprotected – customers often don't use the compromise detection systems operators offer [1–3]. These systems monitor processes, network traffic, and CPU and disk usage from inside the VM or through introspection at the hypervisor to detect if the VM is compromised. We refer to them as OBDs (operator-provided and introspection-based detectors). Customers are reluctant to use OBDs due to privacy and performance concerns. Operators can mandate all 1^{st}-party VMs (those running the provider's workloads) use OBDs but can't require the same of their customers: our measurements of Azure reveal over 95% of VMs don't use OBDs! Operators are thus limited to using non-intrusive methods that prevent VMs from being compromised (e.g., firewalls, ACLs, [4, 5]). But these techniques do not always detect attacks before they succeed (see §2) and without additional protections, VMs in the DC can become and remain compromised for a long time.

It is important to close this gap and protect all VMs. Compromised VMs can be used to attack the DC infrastructure, or another customer's co-located VMs [6]. Operators need to be able to detect compromised VMs and protect all customers without needing their explicit permission or cooperation to do so. Our goal is to provide protection at *DC scale* while *preserving customer privacy*, *without visibility into customer VMs* and *without extensive and expensive monitoring*.

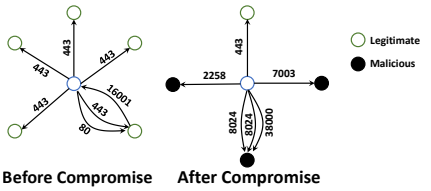

Figure 1: A VM's flows before and after compromise. The numbers on the edges are port numbers.

Here lies an interesting challenge: OBDs monitor process execution, check binaries and VM logins [1–3, 7–12], but without such detailed information, what hope do we have? Two observations help tackle this problem. First, we can learn the behavior of common attackers using information collected from VMs where OBDs *are* deployed. Second, a VM's flow patterns often change once it is compromised [13] e.g., the VM starts communicating with a command and control server (C&C), attempts to find and compromise other vulnerable VMs, or tries to attack the DC infrastructure. For example, Figure 1 shows a compromised VM discovered in Azure and its flow pattern before and after compromise. Our analysis of compromised VMs in Azure shows it is common to observe changes in network flow patterns when VMs are compromised. We expect our observations to be applicable to other providers' DCs as well.

Others have tried using network data to detect compromised machines but their work doesn't satisfy our scalability and privacy needs. Many studies [14–23] route traffic through middleboxes, rely on packet captures, or deep packet inspection (DPI) to extract features which are difficult, if not impossible, to gather through other means (e.g., packet payload). Continuous packet captures at scale come with prohibitive performance overheads and violate privacy by capturing application payloads. Packet captures contain personally identifiable information (PII), e.g. users' IP addresses, with stringent usage requirements [24]. Routing through middleboxes limits scalability, results in single points of failure, adds latency, and reduces throughput [25].

We present PrivateEye, a compromise detection system that runs at DC scale. PrivateEye is tailored to detect common attackers targeting the cloud, runs continuously, and complies with GDPR mandates [24]. It avoids expensive data collection by using flow pattern summaries to detect compromised VMs. It uses OBDs' detections on the VMs the operator can protect to learn the *change* in flow patterns of compromised VMs. Most OBD detections, which we use to train the model, are customer VMs (see §9), and so the learned model is expected to generalize to non-1^{st}-party VMs: PrivateEye applies this model to *all* VMs in the DC. PrivateEye uses random forests (RFs), an interpretable, supervised, machine

learning (ML) model. These models generalize well and can infer, potentially complex, relationships in the input and are interpretable [26]. They often have higher accuracy compared to deployed heuristics (§4) while having similar performance overhead. PrivateEye leverages these properties to offer a practical solution for compromise detection at scale.

PrivateEye's role is to monitor all VMs in the DC and narrow the search space of VMs that need to be investigated. It is designed to have accuracy comparable to OBDs. Once PrivateEye identifies a suspicious VM, operators can use other, more invasive, methods which require customer permission to confirm its detections before shutting down the VM or moving them into a sandbox (see §5).

PrivateEye uses detections from OBDs inside VMs running real workloads to learn the change in the network behavior of compromised VMs. Deployed OBDs (only 5% of VMs) provide PrivateEye with a continuous feed of detections: PrivateEye can identify changes in the attacker's behavior as well as new attacks as it can be continuously retrained in the background. PrivateEye is one of the few systems that can leverage a continuous stream of detections. Attackers may attempt to avoid detection. Retraining may not be sufficient to detect all such attempts, but PrivateEye is still beneficial as it makes it harder for attackers to damage other VMs and the DC infrastructure: they would need to constantly modify their malware to avoid detection. Our contributions are:

1) Creating a scalable, privacy-preserving, compromise detection system that runs without needing customer permission. It operates without packet captures, without DPI, without fine-grained per-packet data, without using IP addresses, and without visibility into the VM.

2) Reporting on a deployment of PrivateEye's collection agent (CA) that has been running on *every* host across all DCs of our cloud for two years. It collects network-level data by querying the vSwitch [27] co-located on the same host.

3) Addressing the practical challenges of extracting features from coarse-grained flow summaries [28]. We use a novel feature construction approach that allows the ML model to detect changes in a VM's flow pattern while protecting customer privacy and keeping the feature vector small.

4) Evaluating PrivateEye using data from our public cloud as well as analyzing the model's false and true positives, feature importance, and design tradeoffs using the same dataset.

Our evaluations on a mix of both our internal and customer VMs, running real workloads, and set aside for testing, shows PrivateEye detects $\sim 96\%$ of the compromised VMs detected by OBDs with only a modest 1% false positive rate. This true/false positive rate is acceptable for our needs.

2 We need DC-scale compromise detection

We first show the need for DC-scale compromise detection:

Cloud VMs are constantly under attack. Brute-force attacks continue to pose a threat to DCs. We show the distribution of the arrival rate of SSH login attempts by unauthorized users to VMs located in 3 major cloud providers and 4 different regions across the globe (Figure 2). We deployed 120 VMs in each provider's DCs. We see VMs are subject to repeated login attempts and at least 3 VMs in each region experience at least one SSH login attempt per second (VMs in these experiments were monitored to ensure none of them are compromised). The time to discovery of a VM – the time from when it is deployed to the first SSH login attempt – is also short: many VMs are discovered in less than 15 minutes (Figure 3). VMs are under constant threat. This is not surprising but it serves to show we need constant monitoring of VMs in case any of these attempts succeed.

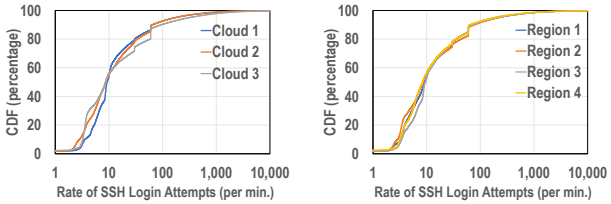

Figure 2: CDF of Rate of SSH login attempts for each provider (left) and for each region (right).

VMs *do* get compromised when customers are careless. Customers may fail to use strong passwords – providers enforce them, but many users change these passwords afterwards. Such VMs are susceptible to brute-forcers. We created 100 of them in our DCs – we ensured they were not co-located with other VMs and monitored them to ensure they did not harm other VMs. We chose passwords from the top 30 of the 1000 most used passwords [29]. The minimum time to compromise – the time from when it was instantiated to when the first successful login occurred – was 5 minutes (password 12345678) with a maximum of 47 hours (password: baseball). These VMs did not have OBDs, and none of them were flagged by any other intrusion detection service.

Compromised VMs are used to attack other VMs. Many exploits require code to be co-located with the victim. Access to a VM in the cloud allows attackers to bypass ACLs and firewalls that only protect VMs from external attackers. Compromised VMs may attempt to compromise other VMs: in one day, our OBDs found 1637 VMs attempting SQL-injection attacks and 74 attempting brute-force login. While small compared to the massive scale of the DC, these numbers only describe those VMs with OBDs. The magnitude of the problem is much greater when scaled up to all VMs.

OBDs (during Jan-June 2018) showed 14% of alerts were VMs brute-forcing other VMs and 13.87% were VMs scan-

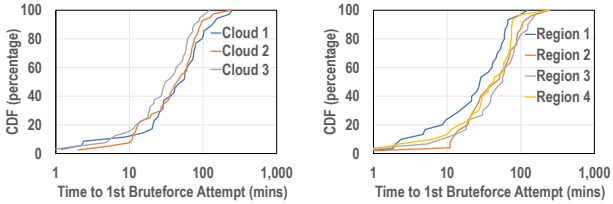

Figure 3: CDF of time between VM deployment and the 1^{st} login attempt per provider (left) and region (right).

	PrivateEye	Heuristic
AUC	0.96	0.5

Table 1: Comparison of PrivateEye to deployed heuristic.

ning for vulnerable ports. Furthermore, 7.7% of compromised VMs were compromised through brute-force attacks.

3 PrivateEye's threat model

Our threat model is similar to other infrastructure services. We do not trust any VM (they can run arbitrary code). We assume we can trust the hypervisors, network, and hardware.

PrivateEye relies on detections from OBDs deployed on a subset of VMs. We use this data as labels during training for the ML model. We assume this data cannot be faked, manipulated, or altered. We also assume OBDs can accurately detect when a VM is compromised. Specifically, we assume false positive/negatives rate of OBDs is sufficiently small – after all, they are accurate enough to be used to protect the provider's 1^{st}-party VMs which are critical to its business.

Malware can adapt its behavior in order to conceal malicious behavior. We assume OBDs adapt to such changes and re-training PrivateEye with their more recent detections can help it adapt to them as well (see §9). We assume many attackers do not discriminate between VMs that are protected by OBDs and those that are not: the same attackers that successfully compromise protected VMs can also (one can argue more easily) compromise others and by learning their behavior, through OBD detections, PrivateEye can protect other (unprotected) VMs in the DC from these attackers.

4 Simple heuristics are ineffective

Our operators have used insights from past OBD detections to build a rule-based solution. To motivate using ML, we compare PrivateEye to this strawman which was used to protect VMs without OBDs in our DCs. It encodes learnings from honeypots and past OBD detections to a per-VM score which measures the similarity of a VM's flows to those attacking honeypots or past OBD detections. Metrics such as having more than 500 flows/sec to a port/IP, changing DNS servers, or too many DNS flows increase the score. The increase is weighted by the operators' confidence in the signature.

This (strawman) heuristic identifies VMs engaged in brute-force or port-sweeping attacks accurately but rarely detects other compromises. We use the area under the receiver operating characteristic curve (ROC) – the graph that plots the true positive rate vs false positive rate of an algorithm – or AUC to measure accuracy. Higher AUCs indicate better accuracy. The heuristic has an AUC of 0.5 (PrivateEye's is 0.96). Indeed, by only testing on port-sweeping VMs, the heuristic's AUC increases to 0.69. Looking at its false positives, 10 legitimate VMs, spanning 3 Virtual Networks (VNets), had scores above 10 (A VNet is a virtual network set up by a user to connect its VMs). Three of the VMs were in our canary VNet. Canaries continuously ping on port 10000 to check network connectivity: all of the VNet's VMs had many flows to port 10000 which is why they all had high scores

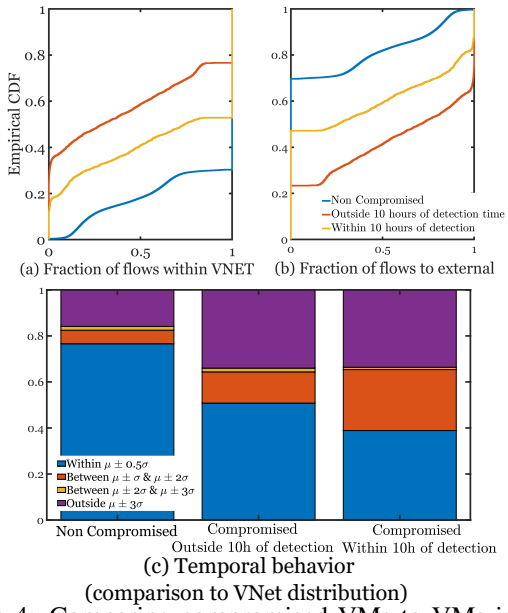

Figure 4: Comparing compromised VMs to VMs in their VNET. μ: average Bps of a VNet to each destination; σ: standard deviation. (a) Distribution of flows to VMs in the VNET. (b) Distribution of flows to IPs outside of the DC. (c) Temporal behavior.

(≥ 2). PrivateEye's high AUC shows using ML with the right features helps avoid such false positives.

5 PrivateEye's design requirements

Our design requirements for PrivateEye are:

GDPR compliance. The European law on data privacy (GDPR [24]) mandates any personally identifiable information (PII) should be tracked in case the customer wishes to inspect (or delete) it. Operators are required to answer customer requests within 48 hours and have two choices: join and tag data with meta-data to be able to identify which customer it relates to or avoid storing any and all PII data. For example, a VM's public IP (and the IPs it communicates with) has to be mapped to the customer's account and stored with *all* network telemetry from their VM. Public IPs are dynamically allocated and would need to be tracked in time which can result in significant and unnecessary overhead.

Even without such customer requests, this data has to be deleted from all company data-stores after 30 days to avoid violations. GDPR makes it expensive to sustain solutions relying on PII data. PrivateEye relies on 10-minute flow pattern summaries and ignores specific IP addresses.

Low runtime overhead. PrivateEye should have low performance overhead, should be able to run at DC-scale, and shouldn't interfere with ongoing traffic. Many, state-of-the-art, compromise detection systems have high performance overhead and cannot be used extensively in the cloud. For example, DPI (e.g., [15]) adds additional per-packet delays which prohibits serving traffic at line-rate ($40 - 100$ Gbps). Packets may be mirrored to dedicated middleboxes that can run DPI off the critical path but our experience with simi-

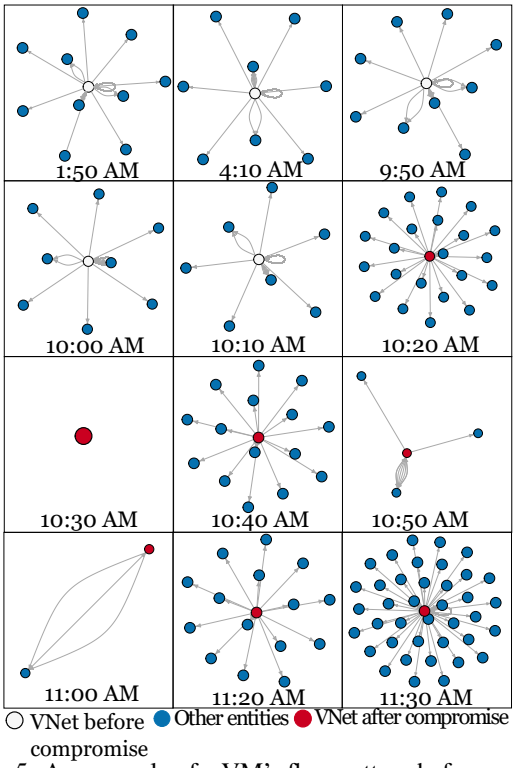

Figure 5: An example of a VM's flow pattern before and after it was compromised.

lar systems (e.g., EverFlow [30]) show they put significant load on the network and that they are hard to scale. We can re-implement DPI based solutions using new programmable switches [31] to improve performance, however, these solutions are not yet ready for production as packets need to be re-circulated [32] which prevents serving packets at line-rate.

Detect malware in the wild. Many past approaches observed malware in a sandbox to build behavioral signatures (e.g., [7]). Malware may change its behavior if it detects it is in a sandbox (e.g, [33]). OBD detections from production VMs provide a rich dataset about malware behavior in the wild which PrivateEye can learn from. PrivateEye uses this learned behavior to detect other instances of compromise.

Ability to generalize. Customers constantly bring up new VMs and shut-down old ones. We should not rely on learning from specific VMs or even customers. PrivateEye can generalize to VMs (and even customers) not in its training set: our evelauation test sets (see 8) comprises of customer and internal VMs and VNets that are never in the training set.

Operate as a first line of defense. OBDs use extensive monitoring. PrivateEye has a more restricted view of the VMs. It is used as a preliminary detector to avoid unnecessary penalty (OBD mandate) on a large number of customers. Its goal is to reduce monitoring overhead and operational complexity and to protect *all* VMs without needing customer permission until further investigation is necessary. Once it flags a VM as suspect, it can raise an alert to the customer to ask for permission to investigate the VMs further through other more invasive and expensive techniques at the operator's disposal.

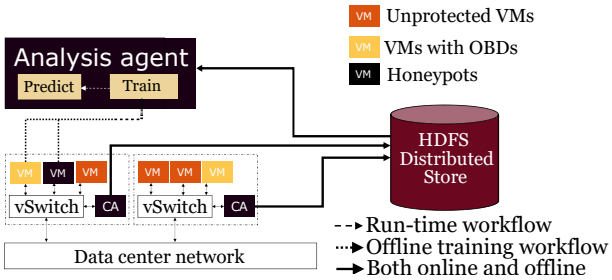

Figure 6: **System overview of PrivateEye.**

For example, our operators have access to VHD-scanners and can also use DPI-based systems on *one-time* packet captures of the VM's traffic[1]. PrivateEye provides "just-cause" for operators to use these tools when it flags a VM as suspect. These approaches are automated and can be run without operator intervention. PrivateEye assigns scores to each detection allowing operators to pick the right tradeoff between the true and false positive rates for their needs. PrivateEye is not meant to fully replace OBDs, we encourage customers who require stronger protections to opt-in to OBDs providers offer.

6 System Design

PrivateEye runs continuously and scales to large clouds with low overhead. The privacy sensitive fields it collects are anonymized using a keyed hash message authentication code (HMAC) during data collection and *deleted* once we construct the features. Figure 6 shows PrivateEye's design. We use two arrow types to differentiate training and run-time workflows. PrivateEye has two parts: the collection agent (CA) and the analysis agent (AA). The CA is responsible for data collection and the AA for analysis and detection. We next describe the key ideas behind its design:

A VM's flow pattern changes when it is compromised. We have observed a VM's flow patterns change once compromised. Almost all malware we studied (using our honeypots) changed the machine's DNS, few connected to the same set of external IPs, some connected on the reserved port for NTP to non-NTP servers, and those mining for digital currency had flows on port 30303. We ask whether these changes are visible on VMs running real workloads (as opposed to idle honeypots)? We leverage 1-month of our OBD detections to answer this question and compare the flow pattern of compromised VMs to others in their VNet. Within each VNet, we see similar behavior for all VMs. But when the VM is compromised, it starts to deviate from the typical behavior of other VMs in its VNet (Figure 4).

Figure 4 a-b compares the *spatial* distribution – the fraction of flows going to other VMs belonging to the same customer vs. to other VMs in the DC vs. machines outside of the DC – of flows originating from VMs sharing a VNet. We observe only 30% of the non-compromised VMs have flows destined to IPs outside of their VNet whereas this number is as high as 50% as we get closer to when the compromise was detected and roughly 80% around the time of detection. It

[1]These are only collected *if* the VM is suspected.

seems, at least in these instances, after the VMs were compromised they tended to communicate more with destinations outside their VNet. The VM's *temporal* behavior also shows changes in behavior around the time of compromise (Figure 4-c). The volume of traffic each VM sends to *individual IPs* is often close to the VNet average but when compromised, VMs exhibit increased deviations from this mean. Our manual analysis of their flow patterns showed noticable changes in behavior after compromise. We omit most of this analysis due to space restrictions but show one example. We show (Figure 5) the flow pattern of a 1^{st}-party VM before it was detected as being compromised (11:30 AM). We see the VM's flow pattern change drastically from its, previously stable, normal behavior when we get closer to the time of detection. We use these insights when constructing features.

We can get accurate flow-summaries from the vSwitch with low overhead. The vSwitch [27] processes all packets of all flows to/from the VM and keeps simple, per-flow state (Table 2) for all active flows. PrivateEye leverages this feature to obtain accurate per-VM flow summaries (see §7.1).

Using OBDs to create labeled data for training. PrivateEye is trained on data from VMs running OBDs. Our DCs run two types of OBDs: (a) Those running on our 1^{st}-party VMs: these VMs often have in-kernel instrumentation for detecting compromise. (b) Those running on customer VMs: customers can opt-in and use OBDs and if they do so, they can deploy and run them inside their VMs. All of the OBDs we use are built on top of Defender [34] (Windows) and ClamAV [35] (Linux) but also monitor irregular login behavior, system calls, and many other parameters. Although OBDs are intrusive the system as a whole meets our privacy requrements: Azure owns all 1st-party VMs, and the only 3rd-party VMs where we use OBDs are those where the customer has explicitly given permission. Note, customer OBDs are less common but, interestingly, despite their lower popularity most compromise detections are from these OBDs (see §9).

We hypothesize many compromised VMs exhibit similar flow-pattern changes to those compromised VMs that were detected by OBDs; because often attackers run attacks against IPs in the cloud irrespective of the services deployed behind those IPs or who owns them (non-targetted attacks §2). Our evaluations confirm this hypothesis as PrivateEye is tested on internal and customer VMs and workloads that are not in the training set and achieves an AUC of 0.96.

Using supervised-learning to learn flow-pattern changes. Anomaly detection and clustering approaches seem natural approaches for solving our problem. We have tried anomaly detection [36], cross entropy [37], ECP [38], TSNE [39], and k-means [40], and also AutoEncoders [41] but anomalies were routinely observed in many VM's lifetimes and it was hard for operators to distinguish between anomalies that were caused by malware and those which were intended VM behavior. Even a tainting + clustering approach i.e., marking points in the clusters with compromised examples as compromised, resulted in higher false positives compared to PrivateEye.

Metric	Description
Time	Timestamp of data collected
Direction	Incoming to/Outgoing from VM
Anonymized Source IP	Source IP in first SYN packet*
Anonymized Source VNetId	VNetId of source IP if any*
Anonymized Dest IP	Destination IP in first SYN packet*
Anonymized Dest VNetId	VNetId of destination IP if any*
Protocol	Protocol if known
Dest Port	Destination port in first SYN packet
BPS In	# of bytes/sec incoming to VM
BPS Out	# of bytes/sec outgoing from VM
PPS In	# of packets/sec incoming to VM
PPS Out	# of packets/sec outgoing to VM
Unique Flows	# of unique 5-tuples collected
Total Flows	# of unique 5-tuples (both collected and missed flows)

Table 2: Data collected by the CA (*not* the features). *These fields are removed entirely after feature creation.

We need to detect *specific changes* that point to the VM becoming compromised (as opposed to finding all anomalies). We chose supervised learning and specifically random forests as they have low overhead. They are debug-able, explainable, highly accurate, and resilient to overfitting [42]. They construct multiple decision trees over a random subset of features during training. Each decision tree uses a greedy algorithm to (1) iteratively pick features with the most information gain [43], (2) makes a decision using values of each feature, and (3) iterates until it reaches a "leaf". Leaves either consist of samples with a single label, or have samples where one label is the majority. At run-time, the algorithm traverses the tree for each test case and returns the majority label (from training) at the leaf. Random forests output the mean prediction across trees and the fraction of trees that predicted each label. We use this fraction as a score to measure confidence and to control PrivateEye's false positives. Operators use it to decide what to do. We set the model's hyperparameters (e.g., max-depth) using Bayesian optimization [44].

Using informative features. In section §7.1 we will describe PrivateEye's CA and how it collects the raw data in Table 2. Privacy-sensitive fields, such as the IPs and VNet ID, are anonymized. Here, we describe the raw data itself, the challenges in extracting privacy-preserving features from this raw data, and how we create these features.

PrivateEye uses three sets of features: graph-based, protocol-based, and aggregate features:

(1) Graph-based features. We want to detect the changes in a VM's flow pattern that indicate it is compromised. The IPs the VM connects to are a crucial part of these flow patterns but using them as features is not possible for two reasons:

Privacy – IPs are anonymized and removed once the features are created. We cannot map IPs geographically nor can we classify them according to the AS that owns them.

Data-sparsity – using IPs as features results in a large feature vector (2^{32}) and by extension an extremely sparse training set. The curse of dimensionality dictates: to maintain accuracy, as the number of features increase, the number of training samples must also increase [45]. This is especially problematic for training supervised models as compromised

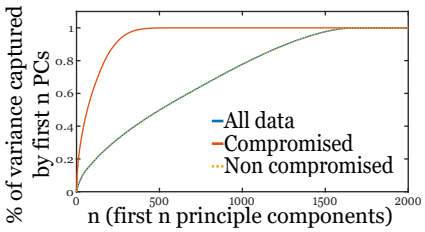

Figure 7: Variance captured by the first n PCs.

VMs are rare – our datasets are imbalanced and have more non-compromised examples and far fewer compromised examples. Using IP-prefixes is not possible: the smallest usable prefix size (to avoid spanning multiple ASes) is $\backslash 24$ but this results in a 2^{24} feature vector which is, again, too large.

Proior work [20, 28, 46, 47] have acknowledged this problem and attempted to solve it by operating at the granularity of flows instead of VMs – classifying individual flows as malicious to avoid using IPs as features. While finding malicous flows is useful when implementing ACLs or firewalls it is hard to identify compromised VMs without viewing their flow patterns as a whole and, in some cases, comparing the VM's flow patterns to that of others. Other works in networking (e.g., [42, 48, 49]) have also used ML. But IPs are not relevant to the problems they tackle and are not used as features. But in the context of compromise detection, IPs play an important part. We present a novel feature construction approach which allows us to summarize graph evolutions, including IP-related changes, using temporal and spatial features. We do not need access to the specific IPs for constructing these features (avoiding privacy problems) nor do we need to use 2^{32} feature-vectors (avoiding data-sparsity problems). Our intuition, based on the observations we presented earlier, is that to detect changes in a VM's flow pattern, features should describe its change both in time and space and compare it not just to its own past behavior but also to others.

Temporal graph-features: capture how a VM's flow patterns change over time compared to itself and other VMs. We will first describe the intuition behind our solution in the context of an example:

Suppose a compromised VM connects to a C&C server with IP a.b.c.d. We can, for all VMs in the DC, build the CDF of bytes sent to IP a.b.c.d over time. If only a few of the other VMs in its DC have been compromised, flows to IP a.b.c.d from the compromised VM would fall in the top portion of this CDF, e.g. the top 10%. The flow to IP a.b.c.d falls into this top 10% interval because such flows were unique to compromised VMs – by mapping the flow to this interval we capture the relevant information for detecting whether the VM is compromised. Other, similar changes are also possible. This is why we use a combination of four such CDFs:

- For each VM, the distribution of each flow according to its "Bps out" over the course of one hour.
- For each VM, the distribution of all flow according to their "Bps out" over the course of 10 minutes.
- For each remote IP, the distribution of all flows to that IP address across all VMs in the DC according to their "Bps out" over 10 minutes.
- For each remote IP, the distribution of all flows connecting to that IP address across all VMs in the DC according to their "Bps out" in 1 hour.

We divide each CDF into five buckets: top 1%, top 1-10%, middle, bottom 1-10%, and bottom 1%. Flows are then projected to a lower dimension based on the bucket they fall into on each of these CDFs . The results are then combined with other features for each VM in each 10 minute period.

These CDFs describe a VM's flows through time as compared to itself and other VMs in the same region. There are other possible CDFs we could use. Finding the optimal selection is the subject of future work.

Spatial graph-features: We classify each flow in one of three categories based on its endpoints: (a) both are in the same VNet (b) both belong to the cloud provider (different VNets) (c) one is an external IP. We aggregate each metric in Table 2 for each group. We can build these groups using the anonymized VNetIds (which we remove after feature construction). Specifically, the same anonymized VNetIds point to VMs in the same VNet, different ones point to VMs in different VNets, and absent Ids point to external IPs.

(2) Protocol Features. A flow's destination port can help identify the application protocol being used. Often attackers/malware use specific protocols for communication. There are numerous examples we found when deploying PrivateEye where the set of protocols used by the VM changed once it was compromised. We created a list of 32 ports of interest including 22 (SSH), 53 (DNS), 80 (HTTP), 123 (NTP), 443 (TLS), 30303 (mining digital currency), and 3389 (RDP). For each of these ports, we aggregated five of the metrics shown in Table 2 to construct six features: "Direction" (incoming vs. outgoing), "PPS In", "PPS Out", "BPS In", "Unique Flows".

(3) Aggregate Features. Finally, we also use the total number of bytes sent/received and the total number of incoming/outgoing connections as additional features.

There is no good linear summary of the feature-set. We have constructed $k = 2116$ features. We next check whether there is a more compact representation of the data using Principle Component Analysis (PCA) [50]. Each principal component (PC) corresponds to an Eigenvalue of the data, and the sum of these Eigenvalues equals its variance. We find we need to keep 75% of the PCs to keep 99% of the variance (Figure 7). Therefore, PCA is not a good candidate for dimensionality reduction in this problem.

Interestingly, if we only focus on compromised VMs, we see 99% of the variance in the dataset can be captured with only 16% of the PCs: the space of compromised VMs (as described by these features) is more compact. But, there is little overall linear dependence across features. This is yet another motivation for using information theoretic and/or ML-based techniques as the number of features is large and it is hard to build human-tuned heuristics using these features.

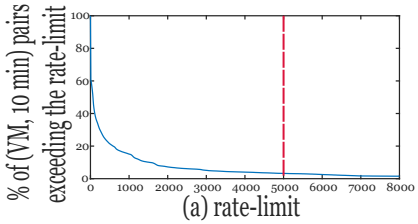

Figure 8: Percentage of (VM, 10 minute) pairs (y-axis) with more than n (x-axis) flows.

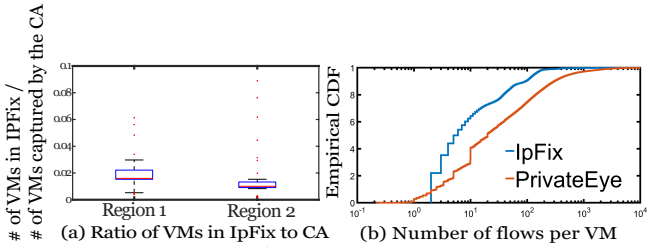

Figure 9: (a) Fraction of VMs captured by the CA also captured by IpFix. (b) CDF of number of flows captured per (VM,10 min) for IpFix vs. the CA (x-axis log scale).

7 System Implementation

We next discuss each of PrivateEye's building blocks in more detail and describe how they are implemented in practice.

7.1 The collection agent

A highly performant CA is crucial in achieving our performance and scalability requirements. We have deployed our CA on all hosts across all our DCs. It runs continuously and polls the vSwitch for the data in Table 2 every 10 seconds. The vSwitch records this data for each flow and for each VM within this period: the choice of polling period does not result in data loss but only impacts CPU usage.

The interface to the vSwitch uses read locks[2]. Although the vSwitch has higher priority to obtain the lock, the CA limits its query to 5000, randomly selected, flows per 10s and per VM to reduce the impact of contention. If the total flows within the polling period exceed this limit for a VM, the vSwitch reports the total. To reduce the overhead of saving data, the CA aggregates some of the fields in Table 2 over 10-minute epochs. This aggregation is on the fields that store bytes, packets, and flows, and we also aggregate flows based on their destination port. As a result, a 10-minute dataset from a region, with over $300,000$ servers, is 109 MB.

To choose the 5,000 limit we looked at the 10s epochs with more flows than any given limit, for one hour, in one DC (Figure 8). Only 4% of samples have more than 5,000 flows. Thus, using this limit results in data-loss for only 4% of the samples. Our data shows on average VMs have 1843.9 ± 9.5 flows in each epoch. The distribution has a long tail; when the number of flows exceeds 5,000 the number of flows is 18890.6 ± 104.0. We accept this loss and show in §8 that we can detect compromised VMs despite this limit. The vSwitch team confirmed the vSwitch continued to process packets at line-rate when using this limit.

We designed the CA from scratch despite systems such as NetFlow [51] and IPFix [52], which are already deployed in our DCs. These systems are used for traffic engineering, DDOS protection, and other tasks. They run on our core routers and sample 1 out of 4096 packets traversing the network core. Because of this sampling, they are biased towards monitoring "chatty" VMs and "elephant" flows. Also, they do not capture flows that do not traverse the network core. Therefore, IpFix/NetFlow are not adequate for PrivateEye which requires more complete knowledge of per-VM flow patterns. Work such as [53] show the shortcomings of such monitoring systems even when used in heavy-hitter detection systems. Other in-network monitoring systems such as [4, 54] are also inadequate as they require a specific type of sampling unsuitable for PrivateEye [4] or require hardware upgrades currently not possible [54]. PrivateEye limits the number of records it extracts in each 10s. This limit also results in occasional data-loss but does not suffer from the same problems. The limit is applied to each VM separately and doesn't bias the dataset towards chatty VMs. We capture data from the host's vSwitch which has all records for the VM's flows irrespective of where they are routed. The CA is a software component on the host and requires no hardware changes. IPFix captures fewer flows per VM than our CA and also misses capturing traffic from a large number of VMs (Figure 9).

The CA has low overhead – typical usage of only 0.1% of the host CPU and a maximum of 15 MB of RAM.

Finally, we note the CA can also be implemented using programmable switches. We use vSwitches as they are more widely deployed in our networks (and those of others).

[2]The table is also used by other systems that need a consistent view.

7.2 The Analysis Agent

The AA has two roles: (1) train a classifier using past detections of OBDs (offline), (2) to run the classifier and predict which VMs are compromised (online). It is alerted when there is a new detection from any of the OBDs on any of the VMs and tags the data from those VMs with a "compromised" label. This data is then added to the training set. This training set is persisted in a distributed data store. The AA is periodically retrained using this data to keep up with any changes in the set of malware or the OBDs themselves.

Data collected by the CA for all other VMs (those without OBDs) is sent through a stream processing pipeline where we create the features. These features are then passed through the RF model which determines whether the VM is compromised. We are in the process of deploying the AA in our DCs. We describe this deployment in more detail in §9.

8 Evaluation

We evaluate PrivateEye using data from Microsoft's cloud. We have shown parts of the evaluation – which helped justify our design choices – in earlier sections (e.g.,§7.1). In this section, our goal is to answer questions about PrivateEye's accuracy (§8.2), the features that help it achieve this accuracy (§8.2), the causes behind its mis-classifications (§8.2), and its performance overhead (§8.3). We also looked into how it can

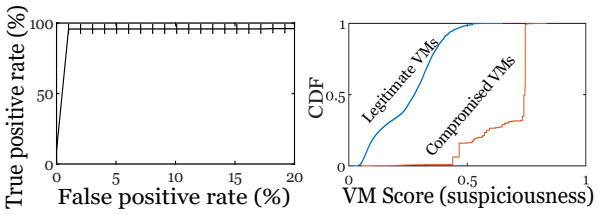

Figure 10: (a) PrivateEye's ROC. (b) PrivateEye's scores (CDF).

be used in the context of an example use-case (Appendix §A).

8.1 Methodology

Data. Each point, (x_i, y_i), in our data corresponds to a VM in a 10 minute period, where x_i is the 2116×1 feature vector described in §6 and y_i is a label: compromised or legitimate.

Labeling. We use OBDs from over 1,000,000 internal and customer VMs to create labeled data for evaluation. These VMs run diverse workloads and use both Windows and Linux OSes. A direct implication of this labeling is that PrivateEye's accuracy can only be as high as OBDs. OBDs can have false positives (negatives), but because operators use them to protect their own 1^{st}-party VMs, we assume these are negligible. OBDs with higher accuracy only improve PrivateEye.

We train and test PrivateEye on *all* detections from these OBDs – we do not restrict PrivateEye to detecting a particular type of compromise: its goal is to detect any compromise OBDs can detect. Most OBD detections were from (different) *customer* VMs (0.87 of the total compromises) – see §9 for further discussion on this. For most OBD detections we also saw external, network-level, signs of malicious behavior. When available, we also report on results from the operator's manual investigations to confirm their detections.

Train-test split. We need to split the data into a train and test set. It is important to do so correctly: if there is *information leakage* between the train and test set we may artificially boost accuracy. For example, we cannot split the data by time because some VMs will have lifetimes spanning both the train and test set. In training, the model may learn the behavior of the VM itself as opposed to whether it is compromised, and when used in practice, it would have much lower accuracy than what we see in testing. The test set should be representative of how the system is used in practice – it is tested on VMs it has never seen before (VMs that are not in the training set). These constraints imply the VMs in the training set cannot be in the test set. We take an even more conservative approach: *We split the data by VNet*. This ensures each VM, and those in its VNet that have similar workloads (§2), only appear in either the training or the test set but not in both.

Addressing class-imbalance. RFs need a similar number of training samples for each label to achieve high accuracy. Unfortunately, our data suffers from *class-imbalance*: a small fraction of our data is labeled compromised (only 0.1%). This can lead to an RF that classifies everything as legitimate while achieving (artificial) high accuracy. We use a standard technique to solve this issue: down-sampling [55]. Down-sampling randomly chooses a subset of the more popular label for training. But too much down-sampling can reduce the volume of data available for training which can also hurt accuracy. We use Bayesian Optimization to find the right trade-off. To improve accuracy, we use the 'balanced' option in SKLearn's RF function which weights the samples for each label to make up for the lower number of samples.

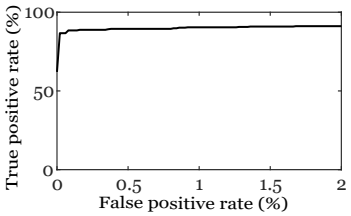

Figure 11: ROC for per VM classification.

Performance metrics. PrivateEye's output for each sample x_i at time t_i is a *score* and a decision on whether the VM was compromised in the 10 minute epoch prior to t_i. The score measures the level of suspicion towards that VM. Operators can use it to decide whether to investigate the VM further.

We use an ROC to measure accuracy – we vary the threshold applied to the scores to decide whether the VM is compromised: ROC shows the true positive rate (TPR) vs the false positive rate (FPR). The area under the ROC (AUC) summarizes the significance of the ROC: an AUC of 1 indicates a perfect test, and an AUC of 0.5 indicates a random test [56].

8.2 Classifier Evaluation

Our goal in this section is to answer questions such as:

- What is PrivateEye's accuracy?
- What causes PrivateEye's false positives/negatives?
- How does PrivateEye's RF compare to other models?
- Are the features we created effective?
- How does PrivateEye's sampling limit affect accuracy?

We next describe the answer to these questions in detail.

Accuracy: PrivateEye has a high AUC (0.96). We use 20-fold cross validation and show the mean of the 20 outputs and the 95^{th} percentile confidence interval (Figure 10-a). PrivateEye can detect 95.77% of compromised samples with 1% FPR. The scores are correlated with the labels (Figure 10-b): the gap between the distribution of scores for compromised and legitimate VMs confirms the accuracy of the model.

PrivateEye's long-term accuracy is also high. PrivateEye checks each VM every 10 minutes to see if any are compromised. But what if operators want to do so less-often e.g., before the VM is shut-down (once over the VM's life-time)? They can aggregate the scores across consecutive samples to do so. Many aggregators are possible: we use the sum of all scores over the lifetime of the VM (other aggregators achieved similar results). The ROC is produced using this new score (Figure 11). PrivateEye's accuracy remains high (90.75% TPR for 1% FPR) albeit slightly lower than before – possibly due to the longer lifespan of legitimate VMs: compromised VMs are typically shut-down more quickly resulting in a lower aggregated score. This result also confirms the ROC in

Figure 10-a is not dominated by samples from a single VM.

We further experimented with different choices of n, where n is the number of 10-minute intervals that need to pass before we make a prediction. The ROC curves were bounded by those of Figure 10-a and b but do not show an explicit trend.

PrivateEye's FNs were *mostly* OBD false positives. PrivateEye's FNR (average) is 5%. This implies (on average) 5% of samples were reported incorrectly as legitimate. But most (70%) of these samples were actually false positives (FPs) of the OBDs (remember OBDs tend to be conservative as they protect internal VMs) – operators investigating the alerts identified them as FPs e.g, in one case the investigation report mentioned: "This was a FP detection as the vulnerability scanner contains data to scan for CVE-2006-3439".

Our investigations of PrivateEye's FNs revealed other interesting information. For example, we grouped FN samples based on which VM they describe and found 86% were cases where all samples for the VM were labeled legitimate. These VMs are part of the training set for other folds during cross validation: it appears PrivateEye is resilient to errors in the labels it uses for training (given the 95% TPR). We need to investigate this point further to confirm this hypothesis.

Some VMs had a mix of compromised and legitimate detections (14% of FN samples). These VMs were involved in port-sweeping attacks but we found no correlation with the number of active flows or the volume of traffic. OBDs assign a "severity" to their detections to report their confidence in the detection. All of PrivateEye's FNs were low severity.

PrivateEye's FPs have similar flow-patterns to compromised VMs. We manually inspect the flow patterns of VMs PrivateEye mistakenly reported as compromised. Most such VMs had a small number of flows to non-reserved ports. In a few cases, the VM had *only* a small number of flows on the DNS port (53) to another VM in its VNet (VMs in the same VNet belong to the same customer). In another instance, the VM had *no* outgoing flows, but multiple VMs from the DC were attempting RDP connections to it. None of the VMs in these VMs' VNets had similar flow patterns. These behaviors are similar to compromised VMs which explains why these VMs were mistakenly flagged by PrivateEye.

PrivateEye detected attacks OBDs missed. We observed two separate instances where SQL servers were conducting port-sweeping attacks on another VM. In both cases the attack lasted for one 10 minute epoch but PrivateEye detected it. OBDs did not. These VMs were only active for a short duration (less than a few hours). The short period of the attack and the short life-span of the VM may explain why the OBDs did not detect these attacks.

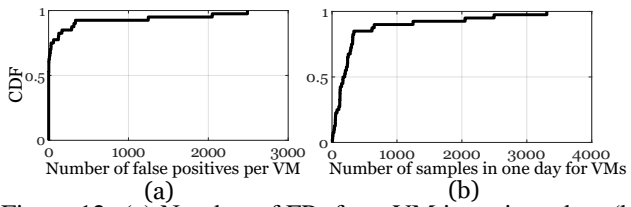

Figure 12: (a) Number of FPs for a VM in a given day. (b) Number of samples in a day for VMs with an FP.

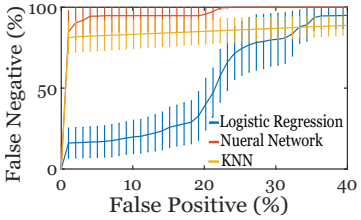

Figure 13: Performance of other ML algorithms.

Consecutive detections can help reduce FPs. We ran samples for legitimate VMs in the test set through PrivateEye and grouped the results by VM. For over 60% of these VMs PrivateEye had only a single FP (Figure 12-a): we could use consecutive detections as a potential means of reducing FPs. For example, we can change the detection granularity to every 20 minutes instead of 10 where we require two consecutive detections to declare a VM compromised. This approach can result in reduced true positives – as seen earlier in the more extreme example where we make one decision in the entire lifetime of the VM. Further understanding of PrivateEye's FPs requires manual inspection from within the VM but, sadly, we are unable to report on the results of such analysis.

Random forests (RFs) are a good first choice [14, 48, 57]. We compared RFs with many other ML algorithms and show RoCs for a subset (Figure 13). RFs outperformed all other models we tried. The closest algorithm to the RF were neural networks[3] (NN) which have an 81% TPR for a 1% FPR.

All features contribute to the detection. We have seen there is no compact, linear, representation of the features that would capture all the information in the data §6. We dig deeper to see which class of features are most helpful. To do so: (1) we use each class individually (Figure 14-a), and (2) we remove each class altogether (Figure 14-b). Removing graph features individually (spatial, temporal) has little impact on TPR (0.2%) but removing both can drastically reduce it (18%). Most classes (except spatial features) can find compromised VMs with a TPR $\geq 70\%$ and an FPR $\leq 5\%$. We conclude all features significantly contribute to detection (though they are not equally important). To validate these observations, we experimented with various feature selection techniques (Figure 15). We refer the reader to [58] for the description of these techniques due to space restrictions.

Aggregate features have good predictive power: a TPR of 77.7% for 5% FPR when used as the only features and resulting in 7% drop in TPR when removed: were most compromised VMs engaging in volumetric attacks? We found this *not to be the case* – the maximum traffic sent by compromised VMs across all samples was 2.6 MBps (median 0.0 Bps) vs. 10.3 GBps (meidan 189.62 Bps) for legitimate VMs.

Comparison to other VMs helps detection. Graph features compare the VM to others by mapping flows onto intervals on various CDFs (see §7.2). What happens if we use CDFs that

[3]We used a single hidden layer with 1000 neurons. Experiments with additional hidden layers in the NN and different numbers of hidden dimensions produced similar results.

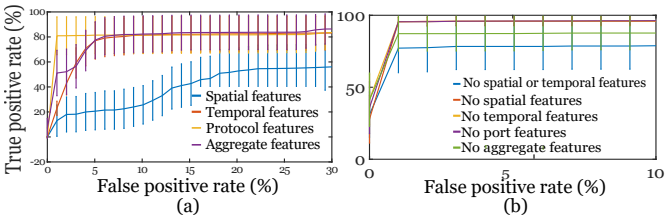

Figure 14: (a) Contribution of each feature class to the overall accuracy. (b) ROC without each feature class.

compare a VM with its own history instead of that of all VMs in the DC? The TPR drops to 80% for 1% FPR (Figure 16) indicating the comparison to other VMs is indeed useful.

The choice of sampling limit is important. The CA rate-limits the number of flow entries it queries from the vSwitch to 5,000 every 10 seconds. This rate-limit allows us to capture over 97% of each VM's flows (§7.1). We next measure the impact of this rate-limit on the TPR by lowering it. We cannot simulate this behavior accurately by "down-sampling" our data as we have aggregated the flows to remove the source port and IP addresses. Instead, we change the rate-limit threshold across all US regions for two months and collect a new dataset. We cannot do a complete sensitivity analysis with multiple rate-limits as this would be costly – we need to capture at least a month's worth of data for training. Therefore, we limit our experiment to just one threshold: 900 flows per 10 second interval. Such a rate limit will result in data loss for over 30% of our training samples (Figure 8). The results show a *significant* decrease in accuracy (Figure 17-a): 80% TPR for 40% FPR. We conclude PrivateEye needs to capture as many flows as possible to maintain high accuracy.

More training data helps compensate for lower rate-limits (Figure 17-b). We start with 92% TPR for 59.6% FPR, and by just adding 8 more days worth of data to the training set it can reach the same TPR for 23.23% FPR.

PrivateEye is unable to detect the type of compromise. PrivateEye cannot specify the type of malware installed on the VM. Our dataset contains additional information about a *fraction* of the compromised VMs e.g., compromised through SSH or RDP brute-force, malware found, port-scanner, port-sweeper, SQL-injection, RDP/SSH brute-forcer, spammer, or others. However, PrivateEye has low accuracy when identifying the type of compromise (70% TPR for 1% FPR) – it is known that multi-class classifiers tend to have lower accuracy [42]. Besides, not all detections have this additional information.

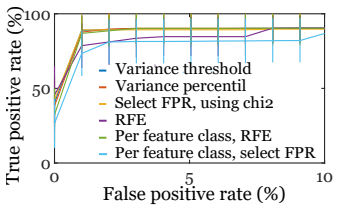

Figure 15: ROC for different feature selection methods.

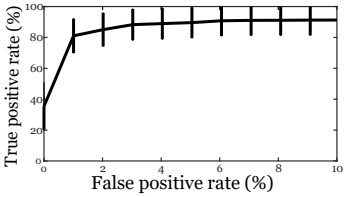

Figure 16: ROC: Impact of the choice of CDFs.

8.3 Performance overhead

One motivation for designing PrivateEye was the need to scale intrusion detection systems to the entire DC. Here we evaluate whether PrivateEye meets this scalability requirement:

- What is the memory and CPU usage of the CA?
- What is the impact of the CA on ongoing traffic?
- What is the expected load on the AA?
- What is the overhead of training the AA?

The CA has low CPU and memory overhead. The CA is deployed across *all* production hosts of a large cloud provider for over 2 years and includes data for over 15,000,000 VMs and 300,000 VNets in the US alone. We chose 100 hosts randomly from DCs across the globe and recorded the CA's CPU and disk usage both in the morning and afternoon. Figure 18 shows its memory usage. Its CPU usage remained bellow 0.1% across all hosts at all times.

The CA does not impact ongoing traffic. Our experience with the CA is that it causes no impact on ongoing traffic. This is in part because the CA has lower priority when obtaining the lock on the vSwitch table. We instrumented the CA on one host to record the time spent, from *user space*, in each query to the vSwitch. The time captures the time spent in contention on the vSwitch table's read-lock and the time it takes to read x entries (where x is the CA rate-limit). There are 8 VMs on the host. We run a SYN flood attack against one of them to simulate different levels of load. We show the time for attacked and non-attacked VMs (Table 3). The results show both the rate-limit and the load (volume of traffic) on a given VM affect the time spent querying the vSwitch for data about that VM but not for data about other VMs. The CPU usage of the CA remained below 0.1% throughout this experiment.

The load on the AA is acceptable. PrivateEye's AA is trained offline using data from the 5% of VMs monitored by the OBDs. The trained model is distributed across each DC region to serve detections every 10 minutes. To quantify the load the AA will have to handle, we looked at the number of flow records per second the CA captured in three regions

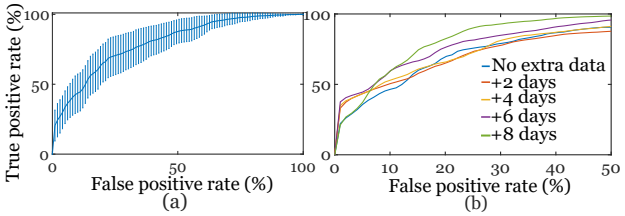

Figure 17: (a) ROC with a rate-limit of 900 flows per 10 seconds. (b) ROC when more data is added.

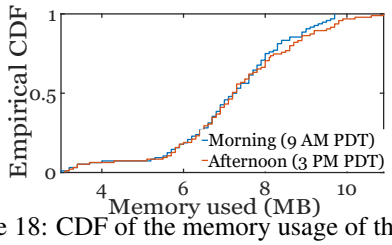

Figure 18: CDF of the memory usage of the CA.

both in the morning and afternoon (Figure 19). The highest rate is around 900,000 flows per second. We expect this volume of data can be easily processed in 10 minute periods.

Training overhead is low. We use Bayesian Optimization to configure our RFs which resulted in an ensemble of 158 trees with a maximum depth of 4. The average training time for such an RF is 5 minutes and $56s \pm 3.76s$ with our two month training set and using 172.87 GB of RAM (peak). The high memory usage is because of the SKlearn implementation which holds the entire dataset in memory.

9 Discussion

This section presents a discussion of the challenges and limitations of a detection system like PrivateEye.

PrivateEye's accuracy. PrivateEye is only as accurate as the OBDs it uses. Thus, our evaluation focuses on comparing PrivateEye to OBDs. OBDs tend to be highly accurate as they are the only systems protecting 1^{st}-party VMs. But, it is challenging to *guarantee* a VM is legitimate. In reality, a legitimate VM is one that passes all detectors deployed in the DC. Should some of these VMs, in fact, be compromised it may cause PrivateEye to have mispredictions (§8). Similarly, we do not know precisely when a VM was compromised but only when it was detected and some of our compromised data may be from when the VM was not yet compromised. Our results in §8 show PrivateEye to be resilient to mislabels.

Generalizing PrivateEye to the entire DC. In our evaluations we partitioned the set of labeled VNets to create a train/test set to emulate how PrivateEye will be used in practice. The VMs we tested PrivateEye on were those which were absent from the training set. These VMs run both Windows and Linux and span a variety of workloads including those of customers who have subscribed to OBDs. We are reasonably confident PrivateEye can detect most compromised VMs. We would have liked to show a small-scale evaluation of PrivateEye where we manually investigated VMs that are

rate-limit	SYN flood flows per second	Query time (μs) for non attacked VMs	Query time (μs) for attacked VM
900	0	585.3 ± 34.1	-
5000	0	2707.27 ± 105.4	-
10000	0	5097.56 ± 261.0	-
900	10000	780.9 ± 110.4	75276.6 ± 6387.5
5000	10000	2933.3 ± 251.7	71961.0 ± 15607.0
10000	10000	5690.1 ± 430.2	75115.5 ± 18360.8
900	50000	504.6 ± 29.2	70760.4 ± 4611.2
5000	50000	2713.4 ± 272.4	75180.8 ± 1639.3
10000	50000	5699.0 ± 289.4	46922.6 ± 9214.63

Table 3: Profiling the impact on vSwitch read-lock. The times are mean across all samples collected over a 1 minute interval.

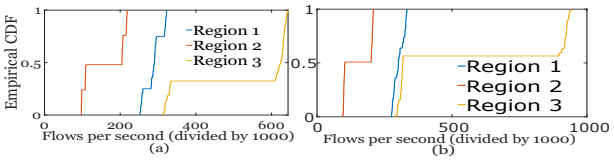

Figure 19: Flows/s captured by the CA. (a) Morning (9 AM-11AM UDT) (b) Afternoon (7PM - 9 PM UDT).

not protected by OBDs. However, we were not able to obtain permission to do so.

Need for retraining. We can retrain PrivateEye to adapt to changes in malware behavior. Retraining may not be enough to allow PrivateEye to detect all such changes, but the change in malware should increase the time to compromise of VMs due to the attackers needing to avoid conspicuous network flows.

The use of ML. Our work on PrivateEye is the first privacy preserving compromise detection system that can run at scale. Other, for example graph theoretic approaches, could be used as well. It is unclear how such algorithms can adapt to changes in malware behavior. This is clearer for ML models where the re-training of the model can update the system. Graph-based NNs are also applicable [59, 60]. It may be possible to improve the accuracy of PrivateEye even further by using these models. This is a subject of future work.

Deploying the AA. We are currently in the process of deploying the AA using Resource Central [61]. Resource Central allows us to store our model in Azure and serves predictions using that model at run-time. It is highly scalable, and we have already used it to deploy several other ML models in production. However, there are still other questions that we still need to answer, for example, who should build the CDFs and what CDFs are best?

Attacks against PrivateEye. PrivateEye itself may be targeted by attackers to reduce the operator's detection capabilities. Adversarial learning [62] is a sub-field of machine learning that studies such attacks. A study of how to guard against such attacks is beyond the scope of this work.

Higher number of 3^{rd}-party compromises. 87% of our compromised data were 3^{rd}-party VMs, however, the majority of monitored VMs in our data are 1^{st}-party VMs. The higher number of 3^{rd}-party compromises is likely due to the tighter protections on 1^{st}-party VMs. We looked at how this could influence our results and conducted preliminary experiments where we eliminated all 1^{st}-party VMs from the data. On average we achieved 86.71% TPR for 3% FPR (83% FPR for 2.5% TPR). We expect accuracy to improve by increasing the number of samples (the dataset has far fewer datapoints than the original) and by re-tuning the model.

Prior work. We have extensively evaluated the performance of PrivateEye and have also compared it to systems currently deployed across the provider's production DCs. Most prior work are not comparable to PrivateEye as they require packet captures (or introspection) we cannot collect because we need customer permission. PrivateEye is not a replacement for

these systems but is designed for DC operators (as opposed to customers). Customers can continue using alternative solutions to protect their VMs. In §7.1 we compared the CA to NetFlow and showed it captures more information. The aggregations and anonymizations applied by the CA prohibit direct comparisons of our AA to NetFlow-based approaches.

Ethical considerations. We annonymized all privacy sensitive information during data collection and *removed them* after feature construction. We conducted all experiments using data from a large cloud provider. The data was either collected from 1^{st}-party VMs under the operator's control or from 3^{rd}-party VMs where the operator had customer permission to monitor the VM. We had explicitly asked permission for deploying the honeypots described in §2 and monitored them closely to ensure they did not cause harm to other VMs. These VMs were not co-located with other VMs.

10 Related Work

We discussed a number of prior works in §1,§7.1, and §7.2. Most do not provide the scalability and privacy characteristics we need [14–20].

Two lines of previous work relate to PrivateEye. One shows the multitude of today's security problems and challenges providers face [6, 63–65]. The other identify malware, compromises, and other types of bad behavior [66–75]. These works can further be divided into two categories:

Network traffic-based Compromise Detection. PrivateEye does not focus on a specific type of attack – it detects any compromise the OBDs can detect. Many prior works identify specific types of bad behavior [5, 5, 9, 12, 19, 21, 22, 47, 57, 76–98]. Some focus on anomaly detection ([99] is a survey of such approaches). Nemean [100] builds intrusion signatures from honeypot packet traces. SNARE finds spammers using packet headers [21]. The work in [18] uses event ordering to identify malware families. VMWall constructs application-aware firewalls aimed at stopping attacks [11]. [90] uses domain knowledge about worms to construct informative features, thus avoiding using IPs (our features capture most of the same information). These works focus on a specific attack which prevents them from comprehensive protection of VMs. Works such as [14–16] rely on packet captures or DPI [101] to identify malicious flows. Packet captures and DPI at DC-scale across all hosts are not possible due to the prohibitive performance overhead. The work of [102] relies on malware propagation to detect the source of attack through analyzing network traffic at key vantage points. However, it does not target identifying the infected nodes. The work of [83] encodes IP addresses through per-source entropies to detect worm attacks; such an approach removes most of the informative properties of individual destination IPs. Such an approach is typically useful when detecting worms and volumetric attacks. The work of [103, 104] discuss other limitations of this approach. Perhaps the closest work to ours is [105] which uses IPFix data from core routers to detect machines that are compromised through SSH brute force attacks. Aside from targeting a specific form of compromise, [105] is based on a fixed set of rules derived through observing a limited set of malware. It is difficult for the approach to adapt to changes in malware behavior. Finally, [47] uses external IP reputation sources to reduce its false positives. This violates our privacy requirements. In addition, we have observed the intersection of malicious IPs reported by commercial IP reputation services and IPs attacking our VMs to be relatively small ($< 10\%$).

Many such works [9, 19, 21, 22, 57] are trained using labels from commercial anti-virus software. Our approach enables us to build a detector that is customized to the cloud because our OBDs detect malicious behavior that occurs in real cloud VMs that are running real workloads. Using OBDs deployed on production VMs running real workloads for labeling allows PrivateEye to avoid problems faced by works such as [106–108] which run malware in emulation mode or in a sandbox to obtain signatures for detection. Many malware can detect when in emulation mode and therefore change their behavior in such situations [18].

Binary-based Compromise Detection. One approach collects malware binaries from honeypots, constructs features from them and then uses Support Vector Machines [7]. Another, clusters binaries found on compromised machines based on their structure, runtime behavior, and the context of the host [8]. Netbait [9] crowd-sources probes gathered from (distributed) infected machines to detect worms. Another approach analyzes memory dumps to construct signatures of the in-memory behavior of malware [10]. Unlike these approaches, PrivateEye performs its classification using networking data alone. Today's privacy requirements, performance constraints, and the new mandates from GDPR make the use of binary and memory inspection techniques in DCs difficult.

Other works also exist [90, 109–113]. Many of these inspired PrivateEye, however, in contrast to these works, PrivateEye's design is aimed at running at scale, having strong privacy requirements, and compliance with GDPR mandates.

11 Conclusion

PrivateEye is a privacy preserving compromise detection system that runs at DC-scale without requiring customer permission. It achieves a true positive rate of 95.77% for a 1% false positive rate.

12 Acknowledgements

The authors would like to thank the anonymous reviewers and our shepherd Hamed Haddadi for their constructive feedback. The authors would also like to thank: Srikanth Kandula, Omid AlipourFard, Vincent Liu, Ricardo Bianchini, Weidong Cui, Ryan Becket, Mina Tahmasbi, Akshay Narayan, Srinivas Narayana, Shadi Noghabi, Robert MacDavid, Ishai Menache, Landon Cox, Hongqiang Liu, and Yibo Zhu for their feedback on our early manuscripts.

References

[1] Microsoft-Inc. Azure security center. https://azure.microsoft.com/en-us/services/security-center/.

[2] Google-Inc. Stackdriver logging. https://cloud.google.com/logging/.

[3] Amazon security solutions. https://aws.amazon.com/mp/scenarios/security/malware/.

[4] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 129–143. ACM, 2016.

[5] Rui Miao, Rahul Potharaju, Minlan Yu, and Navendu Jain. The Dark Menace: Characterizing Network-based Attacks in the Cloud. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, pages 169–182, 2015.

[6] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pages 199–212, 2009.

[7] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. *Learning and Classification of Malware Behavior*, pages 108–125. 2008.

[8] Ahmet Salih Buyukkayhan, Alina Oprea, Zhou Li, and William Robertson. *Lens on the Endpoint: Hunting for Malicious Software Through Endpoint Data Analysis*. Springer International Publishing, 2017.

[9] Brent N Chun, Jason Lee, Hakim Weatherspoon, and Brent N Chun. Netbait: a distributed worm detection service. *Intel Research Berkeley Technical Report IRB-TR-03*, 33, 2003.

[10] Antonio Bianchi, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Blacksheep: Detecting Compromised Hosts in Homogeneous Crowds. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 341–352.

[11] Abhinav Srivastava and Jonathon Giffin. Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 39–58, 2008.

[12] Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. Jackstraws: Picking command and control connections from bot traffic. In *USENIX Security Symposium*, volume 2011. San Francisco, CA, USA, 2011.

[13] Hao Zhang, Danfeng Daphne Yao, and Naren Ramakrishnan. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 39–50. ACM, 2014.

[14] Dmitri Bekerman, Bracha Shapira, Lior Rokach, and Ariel Bar. Unknown malware detection using network traffic classification. In *Communications and Network Security (CNS), 2015 IEEE Conference on*, pages 134–142. IEEE, 2015.

[15] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[16] Michael R Watson, Angelos K Marnerides, Andreas Mauthe, David Hutchison, et al. Malware detection in cloud computing infrastructures. *IEEE Transactions on Dependable and Secure Computing*, 13(2):192–205, 2016.

[17] Fairuz Amalina Narudin, Ali Feizollah, Nor Badrul Anuar, and Abdullah Gani. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 20(1):343–357, 2016.

[18] Aziz Mohaisen, Andrew G West, Allison Mankin, and Omar Alrawi. Chatter: Classifying malware families using system event ordering. In *Communications and Network Security (CNS), 2014 IEEE Conference on*, pages 283–291. IEEE, 2014.

[19] Guofei Gu, Junjie Zhang, and Wenke Lee. Botsniffer: Detecting botnet command and control channels in network traffic. 2008.

[20] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[21] Shuang Hao, Nadeem Ahmed Syed, Nick Feamster, Alexander G. Gray, and Sven Krasser. Detecting Spammers with SNARE: Spatio-temporal Network-level Automatic Reputation Engine. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[22] Guofei Gu, Roberto Perdisci, Junjie Zhang, Wenke Lee, et al. BotMiner: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet

Detection. In *USENIX Security Symposium*, pages 139–154, 2008.

[23] Zainab Abaid, Mohsen Rezvani, and Sanjay Jha. Malwaremonitor: an sdn-based framework for securing large networks. In *Proceedings of the 2014 CoNEXT on Student Workshop*, pages 40–42. ACM, 2014.

[24] General data protection regulation. https://ec.europa.eu/info/law/law-topic/data-protection_en.

[25] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[26] The good, the bad, and the ugly of ml for networked systems. https://www.microsoft.com/en-us/research/video/the-good-the-bad-and-the-ugly-of-ml-for-networked-systems/.

[27] Daniel Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 315–328, 2017.

[28] Xin Li, Fang Bian, Mark Crovella, Christophe Diot, Ramesh Govindan, Gianluca Iannaccone, and Anukool Lakhina. Detection and identification of network anomalies using sketch subspaces. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 147–152. ACM, 2006.

[29] David Wittman. List of commonly used passwords. https://github.com/DavidWittman/wpxmlrpcbrute/blob/master/wordlists/1000-most-common-passwords.txt, 2015.

[30] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 479–491. ACM, 2015.

[31] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[32] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. Fast string searching on pisa. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 21–28. ACM, 2019.

[33] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM workshop on Virtual machine security*, pages 11–22. ACM, 2009.

[34] Windows defender. https://www.microsoft.com/en-us/windows/windows-defender/.

[35] Clam av. https://www.clamav.net/.

[36] Kun-Lun Li, Hou-Kuan Huang, Sheng-Feng Tian, and Wei Xu. Improving one-class svm for anomaly detection. In *Machine Learning and Cybernetics, 2003 International Conference on*, volume 5, pages 3077–3081. IEEE, 2003.

[37] WJRM Priyadarshana and Georgy Sofronov. Multiple break-points detection in array cgh data via the cross-entropy method. *IEEE/ACM transactions on computational biology and bioinformatics*, 12(2):487–498, 2015.

[38] David S Matteson and Nicholas A James. A nonparametric approach for multiple change point analysis of multivariate data. *Journal of the American Statistical Association*, 109(505):334–345, 2014.

[39] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[40] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[41] Geoffrey E Hinton, Alex Krizhevsky, and Sida D Wang. Transforming auto-encoders. In *International Conference on Artificial Neural Networks*, pages 44–51. Springer, 2011.

[42] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 440–453.

[43] John T Kent. Information gain and a general measure of correlation. *Biometrika*, 70(1):163–173, 1983.

[44] Gilles Louppe and Manoj Kumar. Bayesian otimization with skopt. https://scikit-optimize.github.io/notebooks/bayesian-optimization.html, 2016.

[45] Jianping Hua, Zixiang Xiong, James Lowey, Edward Suh, and Edward R Dougherty. Optimal number of features as a function of sample size for various classification rules. *Bioinformatics*, 21(8):1509–1515, 2004.

[46] Chris Fleizach, Michael Liljenstam, Per Johansson, Geoffrey M Voelker, and Andras Mehes. Can you infect me now?: malware propagation in mobile phone networks. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 61–68. ACM, 2007.

[47] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 129–138. ACM, 2012.

[48] Keith Winstein and Hari Balakrishnan. Tcp ex machina: computer-generated congestion control. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 123–134. ACM, 2013.

[49] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the 2017 ACM SIGCOMM Conference*, pages 197–210.

[50] Hans-Peter Deutsch. Principle component analysis. In *Derivatives and Internal Models*, pages 539–547. Springer, 2002.

[51] Benoit Claise. Cisco systems netflow services export version 9. 2004.

[52] Benoit Claise. Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information. 2008.

[53] Jianning Mai, Chen-Nee Chuah, Ashwin Sridharan, Tao Ye, and Hui Zang. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 165–176. ACM, 2006.

[54] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98. ACM, 2017.

[55] Chao Chen, Andy Liaw, Leo Breiman, et al. Using random forest to learn imbalanced data. *University of California, Berkeley*, 110:1–12, 2004.

[56] Area under the curve. http://gim.unmc.edu/dxtests/roc3.htm.

[57] Greg Cusack, Oliver Michel, and Eric Keller. Machine learning-based detection of ransomware using sdn. In *Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 1–6. ACM, 2018.

[58] Feature selection techniques in sklearn. http://scikit-learn.org/stable/modules/feature_selection.html.

[59] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[60] Xavier Bresson and Thomas Laurent. An experimental study of neural networks for variable graphs. 2018.

[61] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167. ACM, 2017.

[62] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

[63] Danny Yuxing Huang, Hitesh Dharmdasani, Sarah Meiklejohn, Vacha Dave, Chris Grier, Damon McCoy, Stefan Savage, Nicholas Weaver, Alex C Snoeren, and Kirill Levchenko. Botcoin: Monetizing stolen cycles. In *NDSS*. Citeseer, 2014.

[64] Hassan Takabi, James BD Joshi, and Gail-Joon Ahn. Security and Privacy Challenges in Cloud Computing Environments. *IEEE Security and Privacy Magazine*, Vol. 8(No. 6):pp. 24–31, 2010.

[65] John P John, Alexander Moshchuk, Steven D Gribble, Arvind Krishnamurthy, et al. Studying spamming botnets using botlab. In *NSDI*, volume 9, pages 291–306, 2009.

[66] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, volume 4, pages 4–4, 2004.

[67] Archit Gupta, Pavan Kuppili, Aditya Akella, and Paul Barford. An empirical study of malware evolution. In *Communication Systems and Networks and Workshops, 2009. COMSNETS 2009. First International*, pages 1–10. IEEE, 2009.

[68] Ting-Kai Huang, Bruno Ribeiro, Harsha V Madhyastha, and Michalis Faloutsos. The socio-monetary incentives of online social network malware campaigns. In *Proceedings of the second ACM conference on Online social networks*, pages 259–270. ACM, 2014.

[69] John P John, Fang Yu, Yinglian Xie, Arvind Krishnamurthy, and Martín Abadi. Heat-seeking honeypots: design and experience. In *Proceedings of the 20th international conference on World wide web*, pages 207–216. ACM, 2011.

[70] Paul Barford and Vinod Yegneswaran. An inside look at botnets. In *Malware detection*, pages 171–191. Springer, 2007.

[71] Vinod Yegneswaran, Paul Barford, and Vern Paxson. Using honeynets for internet situational awareness. In *Proceedings of the Fourth Workshop on Hot Topics in Networks (HotNets IV)*, pages 17–22. Citeseer, 2005.

[72] Vinod Yegneswaran, Paul Barford, and Johannes Ullrich. Internet intrusions: Global characteristics and prevalence. 2002.

[73] Jianxing Chen, Romain Fontugne, Akira Kato, and Kensuke Fukuda. Clustering spam campaigns with fuzzy hashing. In *Proceedings of the AINTEC 2014 on Asian Internet Engineering Conference*, page 66. ACM, 2014.

[74] Seth Hardy, Masashi Crete-Nishihata, Katharine Kleemola, Adam Senft, Byron Sonne, Greg Wiseman, Phillipa Gill, and Ronald J Deibert. Targeted threat index: Characterizing and quantifying politically-motivated targeted malware. In *USENIX Security Symposium*, pages 527–541, 2014.

[75] Luca Invernizzi, Stanislav Miskovic, Ruben Torres, Christopher Kruegel, Sabyasachi Saha, Giovanni Vigna, Sung-Ju Lee, and Marco Mellia. Nazca: Detecting malware distribution in large-scale networks. In *NDSS*, volume 14, pages 23–26, 2014.

[76] Holly Esquivel, Aditya Akella, and Tatsuya Mori. On the effectiveness of ip reputation for spam filtering. In *Communication Systems and Networks (COMSNETS), 2010 Second International Conference on*, pages 1–10. IEEE, 2010.

[77] Christoph Dietzel, Anja Feldmann, and Thomas King. Blackholing at ixps: On the effectiveness of ddos mitigation in the wild. In *International Conference on Passive and Active Network Measurement*, pages 319–332. Springer, 2016.

[78] Alefiya Hussain, John Heidemann, and Christos Papadopoulos. A framework for classifying denial of service attacks. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 99–110. ACM, 2003.

[79] Urbashi Mitra, Antonio Ortega, John Heidemann, and Christos Papadopoulos. Detecting and identifying malware: A new signal processing goal. *IEEE Signal Processing Magazine*, 23(5):107–111, 2006.

[80] Calvin Ardi and John Heidemann. Leveraging controlled information sharing for botnet activity detection. In *Proceedings of the 2018 Workshop on Traffic Measurements for Cybersecurity*, pages 14–20. ACM, 2018.

[81] Katerina Argyraki and David R Cheriton. Scalable network-layer defense against internet bandwidth-flooding attacks. *IEEE/ACM Transactions on Networking (ToN)*, 17(4):1284–1297, 2009.

[82] KyoungSoo Park, Vivek S Pai, Kang-Won Lee, and Seraphin B Calo. Securing web service by automatic robot detection. In *USENIX Annual Technical Conference, General Track*, pages 255–260, 2006.

[83] Arno Wagner and Bernhard Plattner. Entropy based worm and anomaly detection in fast ip networks. In *Enabling Technologies: Infrastructure for Collaborative Enterprise, 2005. 14th IEEE International Workshops on*, pages 172–177. IEEE, 2005.

[84] Mark Allman, Paul Barford, Balachander Krishnamurthy, and Jia Wang. Tracking the role of adversaries in measuring unwanted traffic. *SRUTI*, 6:6–6, 2006.

[85] Paul Barford and Mike Blodgett. Toward botnet mesocosms. *HotBots*, 7:6–6, 2007.

[86] Theophilus Benson and Balakrishnan Chandrasekaran. Sounding the bell for improving internet (of things) security. In *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*, pages 77–82. ACM, 2017.

[87] Mobin Javed and Vern Paxson. Detecting stealthy, distributed ssh brute-forcing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 85–96. ACM, 2013.

[88] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX security symposium*, volume 286. San Diego, CA, 2004.

[89] Ayesha Binte Ashfaq, Maria Joseph Robert, Asma Mumtaz, Muhammad Qasim Ali, Ali Sajjad, and Syed Ali Khayam. A comparative evaluation of anomaly detectors under portscan attacks. In *International Workshop on Recent Advances in Intrusion Detection*, pages 351–371. Springer, 2008.

[90] M Zubair Shafiq, Syed Ali Khayam, and Muddassar Farooq. Improving accuracy of immune-inspired malware detectors by using intelligent features. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 119–126. ACM, 2008.

[91] Ayesha Binte Ashfaq, Zainab Abaid, Maliha Ismail, Muhammad Umar Aslam, Affan A Syed, and Syed Ali Khayam. Diagnosing bot infections using bayesian inference. *Journal of Computer Virology and Hacking Techniques*, 14(1):21–38, 2018.

[92] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. Botgrep: Finding p2p bots with structured graph analysis. In *USENIX Security Symposium*, volume 10, pages 95–110, 2010.

[93] José Jair Santanna, Ricardo de O Schmidt, Daphne Tuncer, Joey de Vries, Lisandro Z Granville, and Aiko Pras. Booter blacklist: Unveiling ddos-for-hire websites. In *Network and Service Management (CNSM), 2016 12th International Conference on*, pages 144–152. IEEE, 2016.

[94] Pavlos Lamprakis, Ruggiero Dargenio, David Gugelmann, Vincent Lenders, Markus Happe, and Laurent Vanbever. Unsupervised detection of apt c&c channels using web request graphs. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 366–387. Springer, 2017.

[95] Hongyu Gao, Jun Hu, Christo Wilson, Zhichun Li, Yan Chen, and Ben Y Zhao. Detecting and characterizing social spam campaigns. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 35–47. ACM, 2010.

[96] Vincentius Martin, Qiang Cao, and Theophilus Benson. Fending off iot-hunting attacks at home networks. In *Proceedings of the 2nd Workshop on Cloud-Assisted Networking*, pages 67–72. ACM, 2017.

[97] Zesheng Chen, Chuanyi Ji, and Paul Barford. Spatial-temporal characteristics of internet malicious sources. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 2306–2314. Citeseer, 2008.

[98] Sakil Barbhuiya, Zafeirios Papazachos, Peter Kilpatrick, and Dimitrios S Nikolopoulos. Rads: Real-time anomaly detection system for cloud data centres. *arXiv preprint arXiv:1811.04481*, 2018.

[99] Monowar H Bhuyan, Dhruba Kumar Bhattacharyya, and Jugal K Kalita. Network anomaly detection: methods, systems and tools. *Ieee communications surveys & tutorials*, 16(1):303–336, 2014.

[100] Vinod Yegneswaran, Jonathon T Giffin, Paul Barford, and Somesh Jha. An architecture for generating semantic aware signatures. In *USENIX Security Symposium*, pages 97–112, 2005.

[101] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. *ACM SIGCOMM Computer Communication Review*, 45(4):213–226, 2015.

[102] Vyas Sekar, Yinglian Xie, David Maltz, Michael Reiter, and Hui Zhang. Toward a framework for internet forensic analysis. In *ACM HotNets-III*, 2004.

[103] George Nychis, Vyas Sekar, David G Andersen, Hyong Kim, and Hui Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 151–156. ACM, 2008.

[104] Mobin Javed, Ayesha Binte Ashfaq, M Zubair Shafiq, and Syed Ali Khayam. On the inefficient use of entropy for anomaly detection. In *RAID*, pages 369–370. Springer, 2009.

[105] Rick Hofstede, Luuk Hendriks, Anna Sperotto, and Aiko Pras. Ssh compromise detection using netflow/ipfix. *ACM SIGCOMM Computer Communication Review*, 44(5):20–26, 2014.

[106] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *USENIX security symposium*, volume 4, pages 351–366, 2009.

[107] Thomas Blasing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *2010 5th International Conference on Malicious and Unwanted Software (MALWARE 2010)*, pages 55–62. IEEE, 2010.

[108] Jaime Devesa, Igor Santos, Xabier Cantero, Yoseba K Penya, and Pablo García Bringas. Automatic behaviour-based analysis and classification system for malware detection. *ICEIS (2)*, 2:395–399, 2010.

[109] Mainack Mondal, Bimal Viswanath, Allen Clement, Peter Druschel, Krishna P Gummadi, Alan Mislove, and Ansley Post. Defending against large-scale crawls in online social networks. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 325–336. ACM, 2012.

[110] Facebook to contact 87 million users affected by data breach. `https://www.theguardian.com//technology//2018//apr//08//facebook-to-contact-the-87-million-users-affected-by-data-breach`.

[111] Bimal Viswanath, Muhammad Ahmad Bashir, Mark Crovella, Saikat Guha, Krishna P Gummadi, Balachander Krishnamurthy, and Alan Mislove. Towards detecting anomalous user behavior in online social networks. In *USENIX Security Symposium*, pages 223–238, 2014.

[112] Stefan Frei, Thomas Duebendorfer, and Bernhard Plattner. Firefox (in) security update dynamics exposed. *ACM SIGCOMM Computer Communication Review*, 39(1):16–22, 2008.

[113] Grant Ho, Aashish Sharma Mobin Javed, Vern Paxson, and David Wagner. Detecting credential spearphishing attacks in enterprise settings. In *Proceedings of the 26rd USENIX Security Symposium (USENIX Security?17)*, pages 469–485, 2017.

[114] `http://heartbleed.com/`.

A Using PrivateEye

PrivateEye is designed as a preliminary detector and its detections should be followed up with more expensive techniques (e.g., [15]). These techniques are computationally expensive and require customer permission. PrivateEye's role is to reduce the number of VMs that need to be investigated and to protect all VMs at all times with low overhead.

Sometimes, obtaining customer permissions takes too long e.g., if a new vulnerability is discovered that could be exploited by compromised machines (e.g., Heartbleed [114]) the provider may not have time to obtain permission. The operator may choose to move suspect VMs to a sandbox[4] until the appropriate patch is applied to all VMs and devices. If obtaining customer permission in time is not possible, PrivateEye can be used to decide whether a VM should be moved or not. But what are the implications of using PrivateEye as the only compromise detection system in the DC?

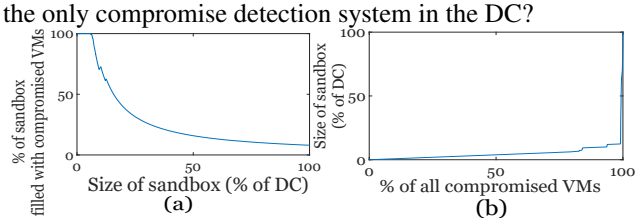

Figure 20: **(a) Fraction of the sandbox occupied by compromised VMs. (b) Sandbox size needed to isolate $x\%$ of compromised VMs.**

For a sandbox size k the operator needs to decide which VMs to put in the sandbox. Using PrivateEye's scores, one choice is to move the top k most suspicious VMs. Figure 20-a illustrates what fraction of the sandbox would be occupied by compromised VMs for different values of k. As the sandbox size increases the utility of the sandbox diminishes–an increasing number of legitimate VMs end up in the sandbox. The choice of k is a tradeoff between the number of VMs that need to be migrated and the number of compromised VMs captured. Figure 20-b examines this tradeoff in our dataset. The operator needs to consider the combination of these graphs when choosing an appropriate k. The larger the sandbox, the more effective it is in reducing the number of compromised VMs outside the sandbox. However, larger k means that it is more likely to place legitimate VMs in the sandbox impacting their performance. To avoid penalizing legitimate VMs, the operator can choose not to migrate a VM if its score is below a threshold. Finding the optimal threshold depends on the operator's needs.

[4] A sandbox could be a host that only runs suspect VMs (limit damage of side-channels) or where more stringent ACLs are imposed.