HideMe: Hiding VMs from Co-Residency Attacks Using Network-Level Traffic Redirection

Bogdan-Nicolae Stănculete
EEMCS
University of Twente
Enschede, the Netherlands
david.bnicolae@gmail.com

Raffaele Sommese
DACS/EEMCS
University of Twente
Enschede, the Netherlands
r.sommese@utwente.nl

***Abstract*—Virtual Machine (VM) co-residency occurs when two virtual machines belonging to different users share the same physical host. Co-residency brings important security implications when one of the VMs is malicious: side-channel leakage, denial of service, and performance degradation are all possible attack vectors that can be leveraged. Achieving co-residency is a two-step process: VM placement and detection. While most of the literature focuses on preventing physical placement, we address the under-explored second step: preventing co-residency confirmation. We present HideMe, a modular, lightweight system that detects malicious probes based on dynamic host behavior and redirects them to decoy VMs. Evaluated in two realistic scenarios, HideMe demonstrates high efficacy, preventing 100% of attacks in consistent traffic environments and 97% in highly variable traffic, all with zero false positives and under strict visibility constraints. As contribution, we also release Hide me under an open-source license, provide deployment instructions for network operators, and outline directions for further improvement.**

I. Introduction

The world is moving at an unprecedented speed. This creates an ever-increasing demand for quick deployment, increased scalability and availability of services. Naturally, the Cloud has emerged as the popular choice for meeting this demand due to its low cost, pay for what you use policy [1], [2]. The rising adoption of Cloud infrastructure often leads to *co-residency*, phenomena in which VMs from different users share the same physical host.

From a security perspective, when a malicious actor achieves co-residency, it gains access to a new category of attack vectors, known as *side-channel attacks* [3], [4]. During these attacks, the adversary exploits the fact that his VM and the victim's share the same hardware (i.e. memory, processor) to steal sensitive information, or disrupt the victim by hindering its access to the shared resources. There are two steps in achieving co-residency:

1. VM Spawning: in which the adversary launches a high number of VMs to maximize the chance that one of them will share the same physical host as the victim.

2. Co-Residency Probing: during which the adversary performs some sort of test to identify whether one of the VMs they launched has been placed on the same physical host as the victim. Most commonly these tests involve *side-channels* [5] and/or probing the victim's public API [6].

While there have been numerous advances in defensive mechanisms aimed at preventing the attacker from co-locating itself with its victim, mainly through better placement strategies [7]–[9] and virtual machine migrations (*proactive* defenses), these approaches are inherently limited. Regardless of how robust the placement strategy may be, the attacker can overcome it by spawning a sufficiently large number of virtual machines. Once co-location is achieved, the defensive mechanism has been rendered useless.

This creates a need for a different array of security measures called *reactive* defenses, which spring into action after the adversary achieves co-location. Despite their complementary nature, little attention has been given to this type of defenses, leaving a gap in the security landscape.

In this paper, we explore this gap by proposing a practical network-level solution that prevents the attacker from confirming co-residency. Our system, HideMe, *identifies* and *thwarts* co-residency tests as they happen, effectively denying the adversary the knowledge that their VM shares the same host as the target. Our main contributions are as follows: (i) we present HideMe: a proof-of-concept solution to identify co-residency probes, using a minimal amount of information. (ii) we present actionable mitigation strategies with limited impact on the availability and performance of the protected service. (iii) we provide the source code and documentation for the proposed solution.

Through this work, we shift the focus from placement-centric to detection-centric mitigation, therefore adding a complementary layer of defense to the existing landscape. Furthermore, we empower both cloud providers and users to reactively protect against co-residency attacks in multi-tenant environments. The solution we propose is modular, configurable and can be seamlessly integrated into existing infrastructure.

II. Related Work

Qiu et al. propose in [10] a novel deployment strategy for VMs that achieves less than 1% probability of achieving VM co-residency. Similarly, Y. Peng et al develop in [8] a different algorithm, based on multi-tenant classification. X. Liang [11] follows the same line of thought and focuses on reducing the chance at achieving co-location. The work of these papers

978-3-903176-75-1 ©2025 IFIP

is on *proactive* defense by designing robust VM placement algorithms. Y. Han [12] even shows that using a pool of algorithms and shifting between them achieves better results for preventing co-residency. We complement their results by creating a defensive mechanism for the scenarios in which the attacker manages to bypass this algorithm and achieve co-location.

On a different note, A. O. F. Atya [13] shifts the focus from proactive to reactive defense, by designing a novel VM migration algorithm. The main limitation in their work is the small scale on which their algorithm was tested (1 victim VM and 2 attack VMs). In a real attack, the adversary would launch tens or even hundreds of VMs, which would exponentially increase the probability that even after migration, the VM is still co-resident with the adversary. We take a different approach, by redirecting the attacker's traffic and denying the very knowledge of co-residency without risking SLA violations from the cloud provider and downtime for the victim.

Miao et al. [9] propose a scheme that reduces the chance of VM co-residency through secure placement and migration of conflicting VMs. Their approach lowers co-residency to 10% on average. However, the main limitation is that an adversary needs only one successful placement to cause harm. In 10% of the cases, the adversary still 'wins'. Our work improves on this by ensuring they have no chance to 'win'.

III. Proposed solution

Our goal is not only to demonstrate feasibility, but also to design a solution for detecting and thwarting co-residency probes that can be naturally integrated into existing cloud environments. This requires a three-step algorithm that: (i) capture the network packets (both legitimate and malicious ones) (ii) determine which hosts are most likely testing for co-residency, and (iii) apply mitigation measures for the identified hosts.

In designing our solution, we chose to align it with the Software Defined Networking (SDN) paradigm, which naturally fits our needs due to the loose coupling between packet monitoring, decision-making, and mitigation. Specifically, data collection and enforcement reside entirely in the *data-plane*, where a programmable switch gathers per-packet metrics, while decisions are handled by the *control-plane*. This design also allows the solution to easily scale to cloud-level demands.

A. Threat Model

When selecting our metrics, we also aimed to identify the minimal amount of information needed to *reliably* detect co-residency probes. To this end, we considered the following constraints in our model:

- Limited visibility of adversary activity: The algorithm can only see the probes addressed to the victim's server and is unaware of any communication between the Command & Control server and the malicious virtual machines. Furthermore, any preparatory steps taken by the adversary prior to probing are also not known.
- Encrypted payload: The payload of the packet is under encryption (e.g. by employing TLS) and cannot be used to distinguish co-residency probes.
- Adversary unawareness: We assume the adversary is unaware of the existence of a filtering mechanism at network level. The assumption is a direct consequence of the limited visibility assumed earlier.

In this limited-visibility scenario, HideMe lacks the information required to link multiple hosts to a single master controller and cannot anticipate which of them will be used for the next probe. If the adversary becomes aware of the filtering algorithm (i.e., the last assumption no longer holds), it can evade detection by distributing probes across a sufficiently large set of hosts. We leave the analysis of less-restricted contexts, with improved visibility and informed adversaries, to future work.

As a consequence of our limited visibility, the only information we can make use of are the number and size of incoming packets, and the metadata present in packet headers. In the next sub-section, we will show how we aggregate that information into three metrics: (i) average number of packets, (ii) average number of connections, and (iii) average packet size.

For simplicity, and without loss of generality, we assume communications use TCP as Layer 4 protocol, as it underlies most common application-layer protocols for public APIs that an adversary might probe. However, our approach is not limited to TCP: for UDP, we can estimate connection counts by inspecting the first packet for identifying header information (e.g., the CONNECT packet in MQTT or the ConnectionID in QUIC) or by counting unique `(SRC IP, DST IP, SRC port, DST port, protocol)` flows.

B. Host Evaluation

In our system, packet collection is performed by a P4 [14] programmable switch in the *data plane*. Beyond regular forwarding, we leverage the switch to collect statistics on bytes and packets sent by each host. These values are then processed by the *control plane* to produce three metrics:

1) **Average Number of TCP Connection Initiations (C)**

$$C = \frac{|\{p \in \mathcal{P} | p.syn = 1 \wedge p.ack = 0\}|}{\Delta t}$$

2) **Average Number of Packets (P)**

$$P = \frac{|\mathcal{P}|}{\Delta t}$$

3) **Average Size of Packets (S)**

$$S = \frac{\sum_{p \in \mathcal{P}} p.size}{|\mathcal{P}|}$$

where $\mathcal{P}$ is the set of all the packets intercepted by the switch which are inbound to the victim's server, and t is time. By *host behavior*, we understand the tuple of metrics

(C, P, S). We then compute the *average host behavior* as a tuple of (G_C, G_P, G_H) where:

$$G_M = \frac{\sum_{h \in \mathcal{H}} M_h}{|\mathcal{H}|} \; \forall M \in \{C, P, S\}$$

and $\mathcal{H}$ is the set of all *distinct* hosts which have sent inbound traffic to the victim's server. We can use the host's behavior to find out its deviation from the average as:

$$\Delta_h = (\Delta_{C,h}, \Delta_{P,h}, \Delta_{S,h})$$

where $\Delta_{M,h} = |1.0 - \frac{M_h}{G_M}|$, and $M \in \{C, P, S\}$ is one of the previously defined metrics. We say a host is *suspect* if its behavior deviation is higher than a given threshold $T = (T_C, T_P, T_S)$ where $T_M \in \mathbb{R} \; \forall \; M \in \{C, P, S\}$. Formally we define the following function for evaluating the host's behavior:

$$Suspect : \mathcal{H} \times \mathbb{R}^3 \rightarrow \{0, 1\}$$

$$Suspect(h, T) = \bigwedge_{M \in \{C,P,S\}} \Delta_{M,h} \geq T_M$$

C. Initiating Mitigation

If a host h is flagged as suspect enough times w.r.t time, that is

$$\sum_{\Delta t} Suspect(h, T) \geq T_{suspect}$$

that host is considered as actively testing for co-residency and HideMe will initiate security measures. This works in reverse as well: if a host (who was previously considered malicious) stops being flagged as suspect enough times w.r.t time, that is

$$\sum_{\Delta t} (1 - Suspect(h, T)) \geq T_{benign}$$

the previously initiated mitigation will no longer be applied to the host. We also enforce that $T_{benign} >> T_{suspect}$[1] to ensure the system responds rapidly to potential threats while preventing oscillating attackers from easily shedding their *malicious* status.

D. Possible Mitigation Measures

We identified two major ways in which we can deny the adversary the information of co-residency:

[1]We say $a >> b$ if $\lim_{a \to \infty} \frac{b}{a} = 0$, that is if $b = o(a)$ as $a \to \infty$.

1) Dropping the adversary's packets: is ineffective because the adversary can immediately tell that the packets are being dropped deliberately by probing the victim's address with a *benign* device. This will prompt the adversary to simply continue probing from a different device and switch again once it is also flagged. However, by dropping the packet, we can be completely certain that the suspect host cannot learn of co-residency.

2) Redirecting the adversary's packets to a different server: is completely invisible to the adversary, and reveals no information about possible defensive mechanisms, as there are multiple commonly used middleware which are responsible (among others) for traffic redirection: load balancers, proxies, etc. Therefore, the adversary will not be prompted (as in the previous scenario) to change devices, as to them everything will look normal. The drawbacks of this approach are the fact that there is a possibility that the server we redirect the traffic to lies on the same physical host, and the need for one or more clone VMs to which we can redirect traffic. The first drawback can be mitigated if the cloud provider keeps an internal mapping of which VMs are deployed to which host. HideMe could then make use of such a map to ensure that the traffic is redirected to a different host.

E. Traffic Adaptation

Another important aspect is that the developed algorithm (in this case the thresholds) must adapt to traffic changes. From the perspective of our algorithm there are three types of changes: *horizontal changes* (the number of hosts fluctuates), `vertical changes` (the hosts' behavior fluctuates) and a combination of the two.

In the first case, no adaptation is needed for the thresholds, as they already model the traffic well enough. If anything, a greater convergence to benign behavior will lead to faster detection of suspect hosts and improved accuracy. To make HideMe adapt quickly to possible `vertical` or diagonal changes, we *reconfigure* the way we aggregate metrics, every time mitigation is triggered for a host, as follows:

1) All the hosts for which mitigation has been triggered are saved in a set called $\mathcal{H}'$, and are excluded from the global average, which in turn becomes:

$$G'_M = \frac{\sum_{h \in \mathcal{H} - \mathcal{H}'} M_h}{|\mathcal{H} - \mathcal{H}'|}$$

2) For the hosts $h' \in \mathcal{H}'$ we consider them as active testers for co-residency unless they change their behavior consistently with the other hosts. More formally, we check if:

$$|1.0 - \frac{M_{h'}}{G'_M}| \approx \frac{\sum_{h \in \mathcal{H} - \mathcal{H}'} \Delta_{M,h}}{|\mathcal{H} - \mathcal{H}'|}$$

If for a host $h' \in \mathcal{H}'$ the above check is fulfilled enough times w.r.t time, that host will be removed from $\mathcal{H}'$,

and its activity will be evaluated according to the original $Suspect(h, T)$ function.

F. Initial Threshold Selection & Multiple Behaviors

Before deployment, we recommend a thorough traffic analysis to determine 2 things: (i) The number of distinct benign behaviors, and (ii) The average deviation for each behavior.

For each type of benign behavior identified, determining the initial threshold $T = (T_C, T_P, T_S)$ can be formulated as a constraint optimization problem, that is:

$$\begin{aligned} &\text{minimize} \quad T = (T_C, T_P, T_S) \\ &\text{subject to} \quad \forall h \in \mathcal{H}, \quad \exists M \in \{C, P, S\}, \quad h_M < T_M \end{aligned}$$

where $\mathcal{H}$ is the set of hosts observed during traffic analysis. For simplicity, the value of T can also be set just above the highest benign deviation (or a little lower) to ensure low false-positive rates. If the benign hosts cannot be put all into one category (e.g., there are different behaviors) our solution can be extended by setting initial global averages for each behavior. This will cause HideMe to first classify hosts based on the average to which their behavior is closest, before starting the normal flow described above.

In this paper, we do not explore multiple host behaviors as our purpose is to quantify the suitability of our solution in the simpler scenario where all benign hosts are assumed to behave similarly, and we leave that to future work.

IV. Testing Methodology

The primary benefit of testing HideMe in a cloud environment is the uncertainty of co-residency, enabling scenarios where an adversary may or may not have a co-resident VM. However, in our context, the presence or absence of a co-resident VM does not affect the algorithm's observations or decisions. Another key feature of the cloud is the availability of multiple hosts communicating over the network, allowing high-throughput traffic generation and address diversification. We argue that this too is non-essential, since high-throughput traffic can be generated locally, albeit with some performance trade-offs, and host diversification can be simulated using spoofing techniques. For stateful connections, we maintain a mapping from spoofed addresses to actual host addresses.

Given that HideMe is intended as a proof-of-concept, and not as a fully-fledged deployment-ready solution, we find that the costs of using a Cloud environment far outweigh the benefits. Therefore, we decided to conduct our testing exclusively in a local environment, which we explain below.

A. The Local Testbed

To emulate the testing network, we used Mininet [15], a lightweight emulator that creates virtual hosts, switches, and links on a single machine. This tool enables reproducible results while preserving the fidelity of real TCP/UDP stacks, routing, and traffic shaping. Since the algorithm's performance depends on host behavior rather than the scale of the environment, we use Mininet to emulate a minimal scenario suitable for our algorithm (see Figure 1).

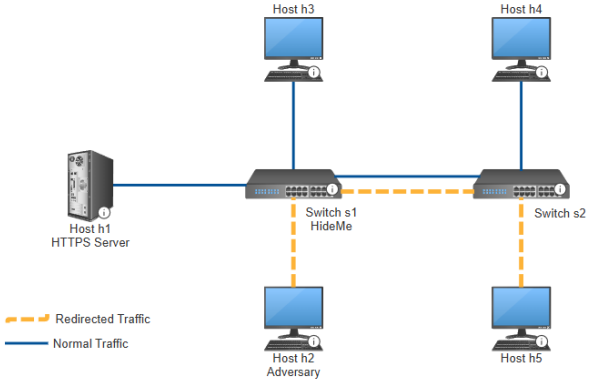

Fig. 1. The Mininet Environment

In this environment, the metric aggregation and decision making is run by the controlling script, while data collection and rule enforcement befalls unto the first switch.

B. Benign Traffic Generation

A key limitation of our work is the representativeness of the traffic used in our simulation compared to the diverse real-world traffic HideMe might observe in deployment. Since the algorithm quantifies behavior, traffic patterns depend heavily on the specific use case. For example, traffic from a High Frequency Trading (HFT) application differs significantly from that of a social-media web service.

To address this, we used publicly available PCAP files containing real-world web application traffic [16] and HTTP request-response pairs [17], replayed via *tcpreplay* (for PCAPs) and *Grafana K6* (for HTTPS traffic). Although this traffic cannot represent every use case, it offers valuable insight into how our solution might behave in realistic scenarios.

C. The Adversary

One of the fastest co-residency detection methods was proposed by Bates et al. [18], achieving detection in under 10 seconds. However, it requires the target and adversary to share the same NIC, a fragile assumption in modern cloud infrastructures. In our testing, we use this time as a conservative upper-bound.

NIC-independent methods, such as the Prime+Probing method proposed by Inci et al. [19], albeit slower, do not rely on this assumption and can be easily applied in public clouds. Still, it detects the presence of *some* co-resident VM, whereas the adversary aims to locate a specific target. In our testing, the adversary implements the memory bus locking technique to detect co-residency, simultaneously probing the victim's public API to confirm co-residency with its target.

Such methods need to be correlated with the probing of the victim's public API in order to attribute the observed results to a specific attack. We argue that a single detection round is insufficient for reliable confirmation. A one-time performance degradation observed at the public API could be attributed to numerous external factors, such as transient

network congestion or high server load, rather than the adversary's actions. Therefore, we conclude that an adversary using similar methods requires at least 2 rounds of detection to reliably confirm co-residency with its target.

Therefore, we say the adversary has conducted a successful test if the probing traffic is not redirected within the first 20 seconds (covering 2 conservative rounds of detection). Accordingly, a test case **passes** if the adversary is unsuccessful, and **fails** if the adversary conducts at least one successful test.

For the test-cases that have not failed (i.e. the adversary is redirected within 20 seconds), we extract the packet counters from the two switches and use it to compute metrics such as *recall* and *accuracy*. All test-cases follow the same structure but differ in duration (taking between 1 and 7 minutes) and traffic model. Generally, shorter test cases aim to identify whether the adversary is successfully filtered out within the allotted time limit, while longer ones observe the long-term behavior of HideMe.

D. Test-Case Scenarios & Performance Metrics

To overcome the fact that our solution is tested in a simulated environment we have opted to use traffic from two real-world examples: HTTP pull requests sent to the Google's Guava repository [17], and anonymized HTTPS traffic captured from a campus network [16]. Our choice in selecting the scenarios is grounded in diversity. Given that the performance of our solution is strictly tied to the traffic model at hand, we wanted to compare said performance in two fundamentally different models. The first scenario presents requests with large payloads, sent in a consistent rate (controllable by us) while the second one presents inconsistent small-payload behavior. Below, we briefly discuss how we model each scenario:

1) The Guava Repository: in this scenario, we generate our own traffic based on the request - response pairs provided, as no PCAP of an actual run is available. We have used Grafana K6 to iterate through the requests in a loop, and had the server reply with the matching response. Due to this, the rate at which pull-requests are made is directly controllable by us (by increasing / decreasing the hosts - *horizontal change*; or the number of virtual users - `vertical change`). Due to the higher degree of control, we have tested both types of changes using this scenario.

2) The Campus Network: The authors in [16] have collected a packet traces captured during seven days of monitoring from eight servers across a large campus network. The traces contain encrypted HTTP traffic over TLS 1.2. For the second scenario we have selected one of the provided PCAPs (corresponding to the last day of observation). Using `tcpreplay` on the hosts, we feed our algorithm with both server and response pairs. One of the key differences between the scenarios is that here we have multiple probes which act as servers. Therefore, our switch was slightly modified to be aware of that. On top of that, we have no control over the rate of the requests as well as the number of distinct IPs. Due to this, we only modeled `vertical changes` in this scenario.

Our results are represented as two-fold: (i) event-level results, where we analyze algorithm decisions such as host redirection and test failures, and (ii) packet-level results, where we aggregate the packets that reached (or not) the target server. All the results are collected at the end of a test-case, and follow the following definitions:

V. Results

In this section we evaluate the effectiveness of HideMe in multiple test-cases covering a wide array of traffic models. In particular, we aim to uncover:

- Does HideMe redirect the adversary within the allotted time limit of 20 seconds?
- Does HideMe maintain high accuracy and recall without causing false positives?
- How robust is the system under changing traffic conditions?

We present a summary of our results across both scenarios and highlight each in detail. We use those results to determine the strengths and weaknesses of our approach.

A. Scenario 1: Guava repository

In Table I we exposed the packet metrics for this scenario. We observe a rate of 0 false-positives with a fast time-to-detect of just 6 seconds on average. Upon further analysis, we have attributed the results to the constant behavior of the benign hosts. We also notice from subtly higher metrics in the case of *horizontal changes* that HideMe benefits from these type of changes to the traffic (both increases and decreases of the number of hosts), as opposed to `vertical changes` which have a more negative impact on the performance.

In terms of event metrics, we have noticed only 1 event (redirection) in all the test cases. The redirection happens within the first 20 seconds, and the host redirected is the adversary. This means that at event level the accuracy of HideMe is 100% (all test cases pass), and the rate of 0 false positives is also maintained.

B. Scenario 2: The campus network

In table II we observe a considerable worse performance than in the first scenario, with a time-to-detect twice as high. The main difference between the two scenarios is that in the case of the Campus network, the traffic pattern is inconsistent: benign hosts come and go. This in turn led to fluctuating global averages, which slowed down the co-residency detection. A surprising result is the fact that the `vertical changes` had no significant impact on the performance compared to the negative impact observed earlier.

In terms of event metrics, we have noticed only 1 event (redirection) in all the test cases. The redirected host is the adversary. However, redirection does not happen within the 20-second limit in all test cases. Overall, the proposed solution achieves a test pass rate of 96.95% in the case of stale traffic, and 97.77% for vertical changes. The highest observed time is 23 seconds, which is just above the imposed limit. These preliminary results are a strong indicative that the proposed

TABLE I
PERFORMANCE METRICS FOR SCENARIO 1. THE SOLUTION DEMONSTRATES FAST DETECTION TIMES AND HIGH ACCURACY WITH ZERO FALSE-POSITIVES IN A STABLE TRAFFIC ENVIRONMENT

Traffic	FPR	FNR	Pass Rate	Accuracy	Time to Redirect
Stale	0%	45.98%	100%	99.26%	6.39 sec.
Horizontal	0%	38.85%	100%	99.62%	6.20 sec.
`Vertical`	0%	49.86%	100%	99.43%	6.55 sec.

TABLE II
PERFORMANCE METRICS FOR SCENARIO 2. HIDEME'S PERFORMANCE IS DEGRADED BY INCONSISTENT BEHAVIOR

Traffic	FPR	FNR	Pass Rate	Accuracy	Time to Redirect
Stale	0%	74.10%	96.95%	97.73%	14.99 sec.
`Vertical`	0%	80.40%	97.77%	97.96%	14.96 sec.

solution has potential for real-world applications, in which the traffic behavior is more chaotic.

C. Key Takeaways

Referring back to the questions posed, the results (albeit affected by artifacts) show that the adversary is successfully redirected, and mitigation measures remain active for the duration of the test case. Therefore, we have achieved 0 false positives at both packet and event level. Consistent behavior proves to be the ideal scenario for HideMe, giving an incredibly fast time of detection. When inconsistent host behavior is introduced, HideMe maintains a test case pass rate of more than 97% with an average time to detect lower than 20 seconds.

The effect of `vertical changes` varies between the two scenarios, and we conclude that we require a third one to be able to predict when a `vertical change` is beneficial and when not. In future works we plan to scale up the number of hosts and test for more complex scenarios where the adversary employs evasive measures.

The number of connection initiations proved to be the key metric to distinguish between co-residency probes and benign traffic in the second scenario, while in the first one that role was played by the average packet size. In scenarios closer to reality, where benign behavior is inconsistent, the average number of connection initiations provides a key discrepancy between probes and benign usage.

VI. CONSIDERATIONS FOR OPERATORS

One of our contributions is releasing the source code of the filtering algorithm. When refactoring the release, we have opted to keep the integration effort to a minimum. Therefore, our code was developed with the following attributes in mind:

- **Hot Reloading**: Changes in HideMe's configuration are detected at runtime, and the filter can re-purpose itself around the new configuration.
- **Modularity**: The source code should be easily integrated in larger, more complex modules.
- **Self-containment**: The source code should be self-contained and should not make assumptions about the underlying packet collection and host mitigation mechanisms.
- **Extensibility**: The filtering algorithm should be easily extensible to support new metrics.

Therefore, we have implemented the filtering algorithm in a event-oriented fashion. The implementation communicates with the rest of the system by publishing and subscribing for events. Since the filtering mechanism is completely decoupled from packet collection and host integration, it can be easily included in any system, regardless of the network paradigm they adhere to (if any) or the choice of technologies.

We provide more details on how to integrate and deploy the solution on our public GitHub[2] repository. Currently, the solution is intended for private and semi-private Clouds where the operator knows the address of the servers intended for protection.

VII. LIMITATIONS & FUTURE DIRECTIONS

First, our solution currently assumes a single behavior pattern among benign users. Therefore, more research is required to evaluate the performance of HideMe in scenarios where hosts exhibit distinct behaviors. Furthermore, the test scenarios can be expanded to evaluate the algorithm's performance under high stress scenarios, where P4 counter overflows or packet drops due to overloads may occur.

Second, the threat model can be expanded to give the adversary knowledge of the existence of a similar filtering mechanism. This will prompt it to employ evasive actions such as distributing the probes and adding random delays between requests. We aim to quantify the impact of such measures and answer if the current solution successfully combats these evasive actions or if adjustments are required.

Third, our tests were conducted in a controlled and simulated environment. Through future research we aim to determine whether there are noticeable differences between our simulations and practical real-world deployment. This includes dealing with overflow errors and reducing computation overhead.

[2]https://github.com/BNStanculete/CoResidency

Finally, through this research we present a lower-bound on the amount of information needed to single-out co-residency tests from benign behavior. Future research could assume more knowledge (usually from the perspective of the network operator) and identify key indicators of such probes which would greatly improve performance and accuracy.

VIII. Conclusion

In this paper we have approached the problem of malicious co-residency in the Cloud from a different perspective. Instead of advocating for better placement strategies (which reduce the odds of co-residency, but can still be beaten through perseverance), we propose a reactive method to deny the adversary the knowledge of co-residency using minimal observable network information.

Our approach is grounded in runtime behavioral analysis and integrates seamlessly with event-driven architectures. While traffic redirection introduces risks such as increased latency, packet loss or SLA violations, our approach achieves a 0% rate of false positives emphasizing its safety and practicality.

Furthermore, the modular and self-contained design allows for compatibility with virtually any infrastructure, including SDN-based, hybrid or even legacy; without requiring major changes. Through this work we aim to enable deployable, low-risk reactive defensive measures against the co-resident threat.

References

[1] D. Lowe and B. Galhotra, "An overview of pricing models for using cloud services with analysis on pay-per-use model," *International Journal of Engineering & Technology*, vol. 7, no. 3.12, 2018.

[2] A. N. Cirlan, "Mining for cost awareness in cloud computing: A study of aws," M.S. thesis, University of Groningen, 2024.

[3] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*, IEEE, 2015.

[4] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *Proceedings of the 11th ACM on Asia conference on computer and communications security*, 2016.

[5] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *2011 IEEE symposium on security and privacy*, IEEE, 2011.

[6] A. Bates, B. Mood, J. Pletcher, H. Pruse, M. Valafar, and K. Butler, "Detecting co-residency with active traffic analysis techniques," in *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, 2012.

[7] A. Srivastava and N. Kumar, "A secure vm placement strategy to defend against co-residence attack in cloud datacentres," *International Journal of Computer Network and Information Security*, vol. 16, no. 2, 2024.

[8] Y. Peng, X. Jiang, S. Wang, Y. Xiang, and L. Xing, "An improved co-resident attack defense strategy based on multi-level tenant classification in public cloud platforms," *Electronics*, vol. 13, no. 16, 2024.

[9] F. Miao, L. Wang, and Z. Wu, "A vm placement based approach to proactively mitigate co-resident attacks in cloud," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, IEEE, 2018.

[10] Y. Qiu, Q. Shen, Y. Luo, C. Li, and Z. Wu, "A secure virtual machine deployment strategy to reduce co-residency in cloud," in *2017 IEEE Trustcom/BigDataSE/ICESS*, IEEE, 2017.

[11] X. Liang, X. Gui, A. Jian, and D. Ren, "Mitigating cloud co-resident attacks via grouping-based virtual machine placement strategy," in *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, 2017.

[12] Y. Han, J. Chan, T. Alpcan, and C. Leckie, "Using virtual machine allocation policies to defend against co-resident attacks in cloud computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 1, 2015.

[13] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. La Porta, P. McDaniel, and L. M. Marvel, "Catch me if you can: A closer look at malicious co-residency on the cloud," *IEEE/ACM Transactions on Networking*, vol. 27, no. 2, 2019.

[14] P. Bosshart, D. Daly, G. Gibb, *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, 2014.

[15] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2010.

[16] S. Špaček, P. Velan, P. Čeleda, and D. Tovarňák, "Encrypted web traffic dataset: Event logs and packet traces," *Data in Brief*, vol. 42, 2022.

[17] T. Bhagya, J. Dietrich, H. Guesgen, and S. Versteeg, "Ghtraffic: A dataset for reproducible research in service-oriented computing," in *2018 IEEE International Conference on Web Services (ICWS)*, IEEE, 2018.

[18] A. Bates, B. Mood, J. Pletcher, H. Pruse, M. Valafar, and K. R. B. Butler, "On detecting co-resident cloud instances using network flow watermarking techniques," *International Journal of Information Security*, vol. 13, no. 2, 2014.

[19] M. S. Inci, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "Co-location detection on the cloud," in *Constructive Side-Channel Analysis and Secure Design (COSADE)*, ser. Lecture Notes in Computer Science, vol. 9689, 2016.