Moving Target Defense for Virtual Network Functions

Reuven Peretz, Shlomo Shenzis, David Hay

School of Engineering and Computer Science, Hebrew University, Jerusalem, Israel.
{reuven.peretz, shlomo.shenzis, dhay}@cs.huji.ac.il

***Abstract*—Network Function Virtualization (NFV) holds a great promise as it provides flexibility and scalability, reduces costs, and promotes innovation (by moving from hardware-based middleboxes to software-based virtual network functions). These benefits, however, expose network functions to security vulnerabilities. In this paper, we investigate two such attack vectors: algorithmic complexity Denial of Service (DoS) attacks and attacks due to co-residency, which include side-channel attacks and DoS attacks on a specific machine. We propose *Moving Target Defense (MTD)* mechanisms—which force an attacker to cope with frequent changes ongoing within the targeted network function to carry out a successful attack through the above-mentioned attack vectors. For algorithmic complexity DoS attacks, we show a mechanism that proactively and reactively switches between different implementations of the network function. Thus, eliminating the certainty of the attacker regarding the targeted implementation. For co-residency attacks, we show a framework to efficiently migrate the virtual network function state without migrating the entire virtual machine, which is prohibitive in such a challenging setting. Our experiments show that both mechanisms can counteract these attack vectors and provide significantly better performance than state-of-the-art solutions.**

I. Introduction

Network Functions (NFs), and more specifically Virtual Network Functions (VNFs), such as firewalls, load balancers, intrusion detection systems, are widely used in networks nowadays to enrich the functionality, flexibility, security, and performance of networks. As such, they are often being targeted for attacks [1]. A common technique used by attackers is to learn and identify the weaknesses of these functions by analyzing their behavior over time and responses to various actions.

Recently, two attack vectors on VNFs have gained attention, both in academia and industry. First, Denial of Service (DoS) attacks, in which a VNF is flooded by adversarial traffic workload, causing its performance to degrade significantly [2], [3]. For example, an adversarial workload may cause the VNF to poorly utilize its cache or cause an underlying algorithm to perform significantly more operations. Another attack vector on VNFs comes from the fact that VNFs share, for a long time, a common hypervisor with other, possibly malicious virtual machines. Such a setting is prone to attacks due to co-residency. For example, side-channel attacks, in which the

978-1-7281-4973-8/20/$31.00 © 2020 IEEE

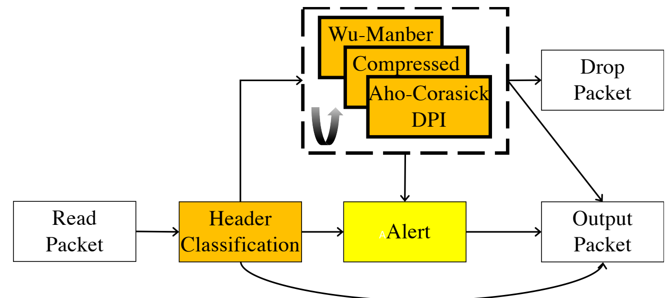


Fig. 1. A processing graph for a simple NIDS. The marked blocks are several implementations of the DPI block, which are reconfigured by the OpenBox controller to provide MTD.

malicious virtual machine can extract (private) information from the network function through either cache or main memory or even OS/hypervisor scheduling artifacts [4]–[8]. Moreover, as the VNF runs on a physical machine, it is prone to traditional DoS attacks on the machine itself (or the communication links to/from the machine).

In this paper, we present *Moving Target Defense (MTD)* techniques to protect VNFs from such attacks. This is done by moving the potential attack targets around the network and possibly changing the targets such that the attacker is unable to learn much about them over time.

We first show how MTD can be used to mitigate algorithmic complexity attacks. To do that, we proactively change the underlying implementation of a VNF, so that a specific adversarial workload will have zero to a small effect on the overall performance. Specifically, this is done using the OpenBox [9] framework, which provides a centralized control plane for VNF logic. The OpenBox controller has a network-wide view of the data plane and the logic it should perform, and thus it is a prime location to manage network-wide defense techniques. Moreover, VNFs are defined in OpenBox as processing graphs of blocks (most of the blocks are generic and common for many contemporary network functions). We have deployed MTD defense mechanisms by adding more implementations of these blocks as well as adding mechanisms to the control plane to proactively (and reactively) switch between implementations of a block without changing the application behavior. See Illustration in Figure 1. We have

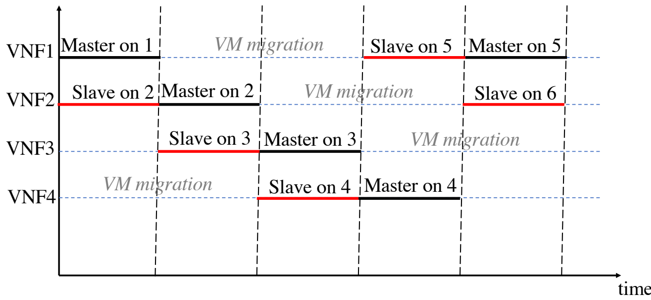


Fig. 2. Illustration of combining active:passive state migration and full-fledged VM migration: 4 VNFs are running on 6 different machines. Until the first switch-over, VNF1 is a master VNF that runs on Machine 1. When the switch-over occurs, VNF2 (running on Machine 2) becomes the master, and VNF1 starts a full-fledged VM migration. When it completes, it starts running as a slave on Machine 5 and keeps updating its state until it is in par with the current master VNF.

demonstrated that such MTD techniques can mitigate known algorithmic complexity attacks, most of which focus on the *Deep Packet Inspection (DPI)* block, which is known to be susceptible to such attacks.

MTD can also be used to mitigate side-channel attacks that are caused by long co-residency of the VNF with another malicious virtual machine (VM) or DoS attacks that are aimed at a specific machine and affects all co-residing VMs. As VNFs are implemented as VMs, a naïve MTD method to tackle this attack vector is to migrate the VNFs periodically, so that one VNF does not co-reside with a malicious VM for prolonged periods. However, such full-fledged VM migration may have poor performance in NFV settings, as traffic cannot be processed while VMs are migrating. Our work builds on recent studies on *fault-tolerant VNFs* [10], [11] and introduces a framework for VNF *state migration*. Specifically, the system uses an active:passive approach for VNF state migration, where a master VNF runs side by side with multiple slave replicas, and a switch-over process occurs periodically after a certain, predetermined time. During the switch-over, one of the slaves becomes a new master, and the master VNF starts acting as a slave VNF, as traffic is diverted to the newly chosen master.

Such a mechanism mitigates many of the attack vectors due to co-residency; yet, for some attack vectors, it is prohibitive that even a slave VNF resides with a malicious VM (e.g., for certain side-channel attack). For these attacks, we have implemented a hybrid approach in which VNFs also perform a full-fledged VM migration *while in slave mode*, achieving the best of both worlds: no VNF resides in the same physical machine for a prolonged time, and there is minimal interruption to the VNF performance as migration happens in the background. See Illustration in Figure 2.

This paper is organized as follows. In Section II, we present related works that discuss either NFV vulnerabilities or relevant MTD techniques. Section III presents our MTD framework to mitigate algorithmic complexity attacks on VNFs, while Section IV presents our MTD framework to mitigate side-channel attack due to the co-residency of VNFs with other (possibly malicious) virtual machines. Concluding remarks are given in Section V.

II. Related Work

Moving Target Defense (MTD) is a defensive strategy that aims to reduce the asymmetry in cyber attack-defense confrontation. There are many MTD techniques, tailored for different scenarios and various settings [12]–[27]. However, to the best of our knowledge, no MTD technique was geared directly to NFV.

Perhaps the closest existing MTD framework for network protection, taking advantage of algorithmic diversity, is *Resilient Dynamic Data Driven Application Systems* (rDDDAS) [28]. rDDDAS consists of several modules, including *Software Behavior Encryption* (SBE), *Decision Support System* (DSS), and *Autonomic Management* (AM). In a nutshell, under rDDDAS, the application run time is divided into phases, where the output of each phase is the input of the next one. Several functionally-equivalent, however, behaviourally-different replicas are run in parallel. The DSS module evaluates the state of each replica at the end of each phase in terms of solution range, memory utilization of the program, and the variable values of subsequent iterations diversion. Then, the DSS determines which of the replicas' output to select as an input for the next phase. The output state of the replica that performed the maximum number of iterations is selected for the next phase. The Autonomic Management (AM) module is responsible for the resiliency configuration, i.e. when to shuffle the current variant, the shuffling frequency, and the variant selection for the next shuffle. As well as using SBE, the framework uses replication (hardware and VM redundancy) and automatic check-pointing (using CPPC [29]). The authors focus on the protection of application executing nodes on the network. In their examples, they provide a MapReduce application for word count in a large text file and a Jacobi based Linear Equation Solver. Such applications may suffer performance degradation or crashes due to Denial of Service (DoS) attacks or explicit attacks of the application run time. Moreover, as rDDDAS is computationally-intensive, requiring running several modules in parallel, it is unlikely it will fit NFV settings.

MTD method for Information leakage across co-resident VMs was studied in [4], which introduced Nomad—a migration-as-a-service cloud computing service. Nomad uses previous VM placements to decide the next placement action, intending to reduce the information leakage across arbitrary pairs of clients. Nomad uses live migration in every placement action and deals with general virtual machines, and not specifically VNFs. We show that, in the NFV setting, state migration can be done more efficiently as full-fledged live migration is not always needed and propose a lightweight mechanism for it.

As for the attack vectors we consider in this paper, Pedrosa et al. [3] propose *Cycle Approximating Symbolic Timing*

Analysis for Network Functions (CASTAN), a tool that automatically synthesizes adversarial workloads for NFs. Given the LLVM code of an NF and a processor-specific cache model, CASTAN tries to discover execution paths that consume relatively large numbers of CPU cycles and synthesizes workloads that trigger them. Their results show that under ideal circumstances, a CASTAN workload is able to increase NF latency by 201% and decrease throughput by 19% when compared to typical test network traffic. For specific NFs, mostly network intrusion detection systems (IDS), earlier works have built a worst-case workload that could degrade their performance even more (e.g., against Bro IDS [30] and Snort IDS [2], [31], [32]). These workloads target the *Deep Packet Inspection* component of the IDS, which is the component that consumes the most resources in the first place.

Steinberger et al. [33] have investigated the effectiveness of MTD in the context of Software Defined Networking at the ISP level. They have shown that employing both network-level MTD and host-level MTD can reduce the overall probability of a successful DDoS attack. Furthermore, as the level of collaboration between ISPs (and the deployment of MTD techniques) increases, the probability of a successful attack decreases. Unlike our work, the MTD methods that were considered in [33] are changing traffic *routes*, which do not mitigate the attack vectors against NFV we consider in this paper.

III. MTD FOR ALGORITHMIC COMPLEXITY ATTACKS

Our MTD approach to mitigate algorithmic complexity attacks is to employ N-version programming to create a constantly moving, unpredictable attack surface for an adversary. Specifically, we create several functionally-equivalent versions of the same application, where the underlying algorithmic implementation of each version is different. To deploy and control these versions (as well as to switch between versions), we use the OpenBox [9] framework. OpenBox provides a clear separation between NFV control-plane and NFV data-plane, as well as inherent support for NFV modularity. This comes handy in our case, as we can define a functionally-equivalent version for each such software module (which we call *block*), and not for the entire VNF, and extend OpenBox to be able to switch between different implementations of the same block.

Specifically, OpenBox's data-plane is built around servers (which are called OpenBox Service Instances or OBSIs) that run (one or many) network functions assigned for each of them by the control-plane (which is called OpenBox controller or OBC). When an OBI is deployed, many software blocks are installed on the OBI (where additional blocks can be installed on-the-fly). The OBC instructs the OBI which of these blocks to run, by sending the latter a *processing graph* that defines the relations between the blocks as well as the block configurations (recall Figure 1). Thus, in our MTD, we have provided several functionally-equivalent versions of the same block incorporated into an OpenBox application, and extend the OBC to support switch-over between versions.

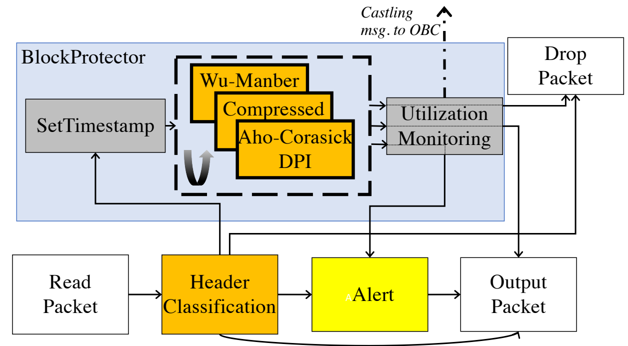


Fig. 3. The processing graph for a simple NIDS, as described in Figure 1, with the additional components (shaded in gray) used for measuring performance and react in real-time to algorithmic complexity DDoS attacks.

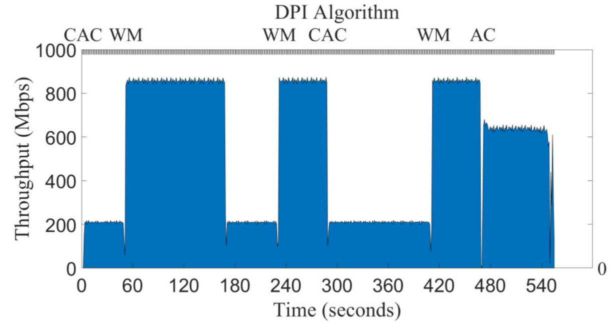


Fig. 4. The throughput of our sample VNF under normal traffic, when the OBC proactively changes the underlying DPI algorithm.

Specifically, our algorithmic complexity attack MTD extension consists of both *proactive* and *reactive* attack mitigation techniques. For the proactive mechanism, the OBC maintains a timer, which upon timeout, forces a random subset of the currently connected OBSIs to switch to newly generated processing graphs. Although the new graphs consist of the same application logic, the application variants are randomly chosen from the user-provided repertoire. The timeout is also skewed by a random interval to increase the complexity for the attacker. For the reactive mechanism, we have devised a new OpenBox entity, which we call BLOCKPROTECTOR (see Figure 3). This entity takes a user-provided subgraph (namely, one block or more) and wraps it. A Timestamp is added to each packet arriving at the input of the protected subgraph. The packet then enters the user subgraph. On the output of the subgraph, the packet enters into a utilization monitoring element, which measures the latency which was imposed by the user subgraph for the current packet. If the current latency (averaged on a certain sliding window and compared with deviations in the past) increases above a carefully-chosen threshold factor, a *castling* message is sent to the OBC containing the name of the protected block emitting the alert. Then, the OBC, based on a global view of the network, might decide it is caused by an attack on the OBSI, and reinstall a new processing graph to mitigate the attack.

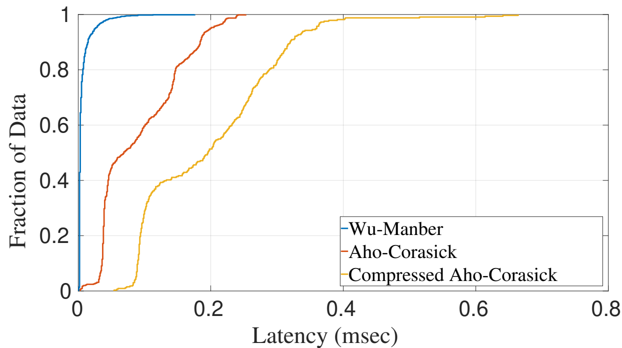


Fig. 5. Empirical CDF of Wu-Manber, Aho-Corasick, and Compressed Aho-Crasick under normal traffic.

A. Evaluation

We have implemented our OpenBox control-plane extension as well as the corresponding OBSI [1]. Our OBSIs use FastClick execution engine [34], which processes packets in batches. In order to support this mode of operation, we added batching support to all our custom blocks.[2] Furthermore, as FastClick also supports NetMap [36] and DPDK [37] for kernel network stack bypass, we have added these capabilities also to OpenBox.

As in previous research [2], we have focused our evaluation on the deep packet inspection block, and considered three functionally-equivalent DPI blocks, each with a different strings matching algorithm: Aho-Corasick [38], Compressed Aho-Corasick [2], and Wu-Manber [39]. While each of these algorithms is vulnerable to algorithmic complexity attack, the adversarial traffic for each of them is different: for example, adversarial traffic for Wu-Manber does not have any effect on Aho-Corasick algorithm and vice versa. We have used a simple network function, whose functionality is similar to a network intrusion detection system that examines both packet headers and packet payload. This network function was defined using three processing graphs, each using a different DPI block. The string matching algorithm was configured with 10,000 rules extracted from the ClamAV database [40].

All tests were deployed on a machine featuring an i7 2600K CPU, 12GB DDR3 physical memory and a SATA2 500GB disc and running Ubuntu 16.04. The traffic generator machine was running on a separate physical machine. Both the OBC and OBSI were deployed on the same system provided with all available resources. When testing, the maximum throughput of the link between the traffic generator and the OBSI admitted average rates of 961 Mbps. A packet trace consisting of 130K packets from crawling the 80 top visited websites according to Alexa, as well as traces available from FIRST 2015 conference [41], were replayed from the traffic generator to the OBSI. Using `ifstat` on both the input and output ports of the OBSI, the ingress and egress rates were measured. Figure 4 shows the throughput of our sample VNF over time, as the OBC proactively changes the processing graph, and at each epoch uses a different string matching algorithm. As expected, under normal traffic Wu-Manber and Aho-Corasick algorithms outperform the compressed Aho-Corasick algorithm. When switching between algorithms (namely, reconfiguring the OBSI), the OBSI experiences short downtime of 0.8184–1.405503 seconds for Wu-Manber and compressed Aho-Corasick algorithms [3], while configuring the Aho-Corasick algorithm takes up to 3.5 seconds as its data structures are significantly larger. Thus, the OBC must make sure proactive reconfigurations do not occur too often and are not predictable by attackers (that may devise an adversarial workload for the current running algorithm). In our implementation, this is done by randomly deciding (for each OBSI separately) every 60 seconds whether to perform a reconfiguration or wait another 60 seconds.

Although our proactive approach makes it more difficult for an attacker to devise the suitable adversarial traffic, we take extra precaution by adding a reactive component to our MTD mechanisms: upon detection of a deviation of the expected latency of packets under a specific algorithm, the OBC is notified and reconfigures the OBSI to use a different algorithm immediately. The threshold is set by ongoing measurement of packet latency on normal traffic. If the average latency of an algorithm over a window of 10000 packets (and up to 5 seconds) is above the 5th percentile of the latency, then reconfiguration is initiated. The next algorithm is chosen randomly between the remaining two. Figure 5 shows the CDF of the latency of the three algorithms, showing that the latency threshold is at least twice the average latency.

In order to evaluate the performance of our reactive mechanism, we have devised adversarial traffic for each algorithm and sent it to the VNF to see the performance hit. Figure 6 shows the throughput hit by the attacks, and how the OBSI recover from a hit when it is reconfigured to a different algorithm. Figure 7 shows how attacks affect the latency of packets and when the reactive actions are initiated.

IV. MTD for Attacks Due to Co-Residency

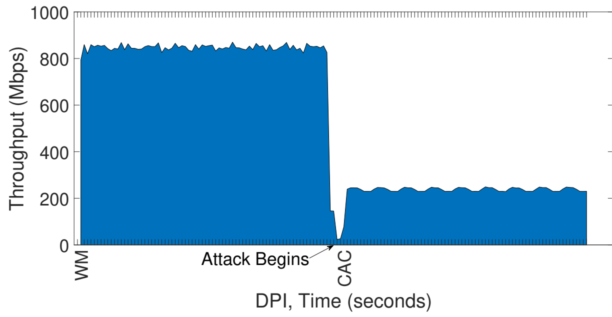
As these attacks happen when the attacking VM and the victim VNF share the same hypervisor (namely, they reside on the same physical machine), a naïve and costly MTD solution, is to periodically migrate the VNF to different physical machines, and steer the traffic destined for that VNF to the new location. With such migration, the attaching VM and the VNF will not share the same physical machine for a long time, stopping these side-channel attacks. Moving VNFs around are useful for other attack vectors, such as compromising the hypervisor itself or launching a targeted Denial of Service on a specific physical machine in the network (or its corresponding links).

A major disadvantage in performing VNF migration is the resulting downtime of the machines, implying that a huge amount of traffic is either dropped or need to be buffered

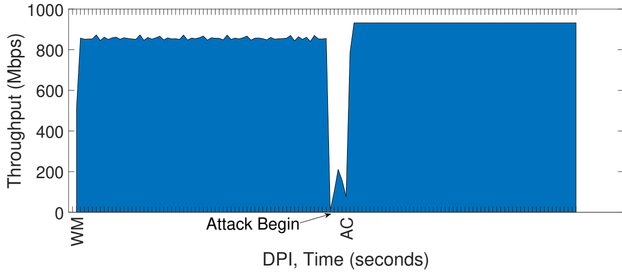
[1]Implementation is available in `github.com/shlomos/OpenBox/-tree/mtd`.

[2]The original OpenBox's OBSIs use Click execution engine [35], and therefore, achieve significantly worse performance.

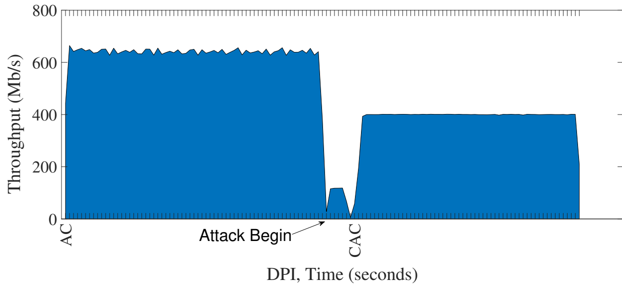
[3]The exact numbers depend on the networking stack used.



(a) Attack on Wu-Manber algorithm, switching over to compressed Aho-Corasick algorithm.

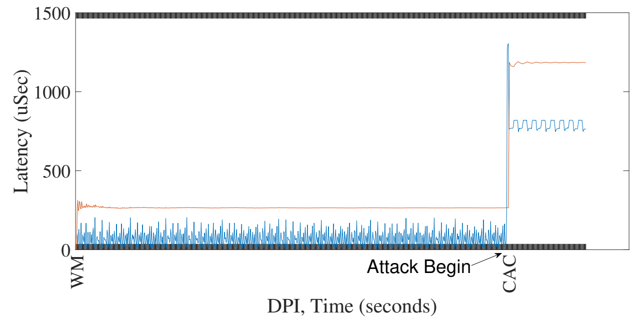


(b) Attack on Wu-Manber algorithm, switching over to (classic) Aho-Corasick algorithm.

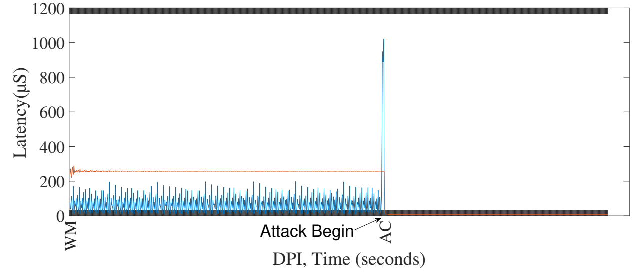


(c) Attack on (classic) Aho-Corasick algorithm, switching over to compressed Aho-Corasick algorithm.

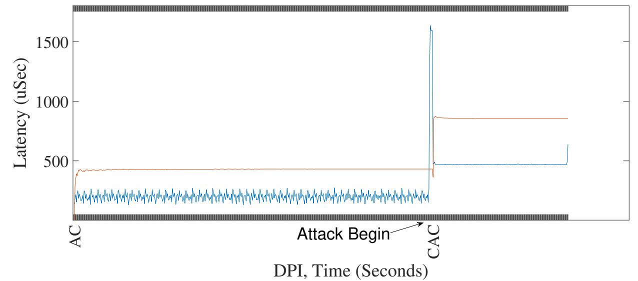
Fig. 6. The changes in throughput due to a successful algorithmic complexity attack on the running DPI algorithm, and OBSI recovery after switching over to a different algorithm.



(a) Attack on Wu-Manber algorithm, switching over to compressed Aho-Corasick algorithm.



(b) Attack on Wu-Manber algorithm, switching over to (classic) Aho-Corasick algorithm.



(c) Attack on (classic) Aho-Corasick algorithm, switching over to compressed Aho-Corasick algorithm.

Fig. 7. The changes in latency due to a successful algorithmic complexity attack on the running DPI algorithm, and OBSI recovery after switching over to a different algorithm. Blue line denotes the latency experienced by packets, while the red line denotes the latency threshold used by our algorithm.

somewhere in the network for post-migration processing. In our MTD mechanism, on the other hand, we propose to migrate only the *VNF state* and to combine it with an "active:passive" approach to shorten the downtime of VNFs. This approach is inspired by recent efforts to provide fault-tolerance to VNFs [10], [11], where VNFs need to survive a failure without losing their state.

Specifically, in our system, a master VNF runs side by side with multiple slave copies, and a *switch-over process* occurs periodically, where one of the slaves is turning to the new master, and the master VNF starts acting as a slave VNF.

As FTMB [10] and FTvNF [11], our framework consists of several components as described in Figure 8: Packets arriving at a master VNF first go through a packet logger component. The packet logger generates a unique ID for each packet, adds it to the packet, and stores the packet in persistent storage. In our implementation, as in [11], we push a 36-bit packet ID

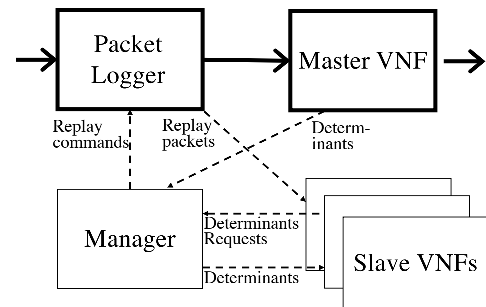


Fig. 8. Block diagram of the component of our MTD framework for attacks due to co-residency. The framework's datapath consists of the blocks marked in bold. The master VNF produces determinants and send them to the manager. The manager instructs the packet logger to replay packets to specific slave VNFs by issuing replay commands. Upon receiving a packet from the packet logger, a slave VNF requests the corresponding determinants from the manager.

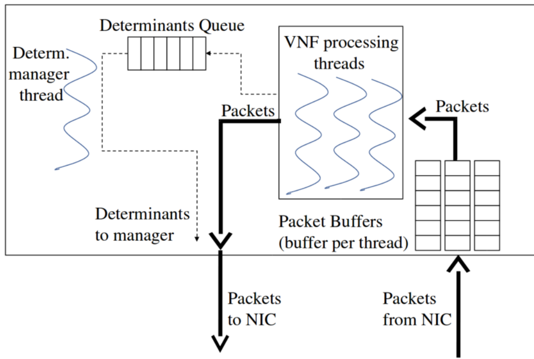


Fig. 9. Illustration of our VNF operation, when it acts as a master. Packets are received from the NIC, processed by the VNF threads, and sent to NIC towards their next destination. While processing packets, determinants are generated and stored in the determinants queue, where they are evicted and sent to the manager in batches by the determinants manager thread. Packets sent to the master VNF immediately after the switch-over operation and before the VNF managed to restore the correct state, are stored in packet buffers and processed after the state is restored. After a buffer is drained, it stays empty for the rest of the VNF's operations as a master.

as three VLAN tags. From the packet logger, packets are sent through to the master VNF. All packets are stored until they are fully processed by all VNFs (master and slave copies). A manager component observes the packet progress in an asynchronous manner; when the master VNF is done with some packet p, the manager instructs the packet logger to send p to all corresponding slaves; packet p is deleted from the packet logger when the master VNF and all slaves finished processing it.

One of the significant challenges of successfully migrating a network function state is to handle non-deterministic state changes correctly, such as time-based variables and random processes. As in [10], [11], we carefully log packet access data (called *determinants*), generated each time a VNF access a shared variable, or call a function that uses hardware or non-deterministic operation. These determinants are recorded by the master VNF in the manager. Slave VNFs then asynchronously use these recorded values (instead of performing the non-deterministic operation), and therefore, can accurately emulate the master VNF's behavior. Upon a switch-over, the designated slave needs to process only the remaining packets (and determinants) in order to recover the correct state, assume the role of a master, and start processing new packets (and determinants). New packets that arrive during the switch-over process, are stored in packet buffers within the designated VNF. These packets will be processed upon completion of the switch-over process until all buffers become empty. See illustration in Figure 9.

It is important to notice that some of the attack vectors (for example, several side-channel attacks) are not mitigated even if the malicious VM co-resides with a slave VNF for a prolonged time. For such an attack vector, a full-fledged VM migration is needed. Unlike previous works (e.g., [4]), our framework allows such a migration to happen in the background, without resulting in any performance degradation.

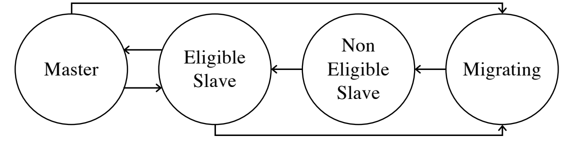


Fig. 10. State diagram for a VNF in our framework.

TABLE I
REPRESENTATIVE MIGRATION AND SWITCH-OVER TIMES IN OUR FRAMEWORK.

	Average (msec)	Maximum (msec)	Minimum (msec)
Migration time	14853.681	16599.016	13207.868
Time as non-eligible slave	18357.818	20492.338	16328.589
Switch-over time	144.007	208.799	67.578
Buffer draining time	57.258	78.696	38.648

For example, Figure 2 shows a possible setting combining both our active:passive state migration and full-fledged VM migration.

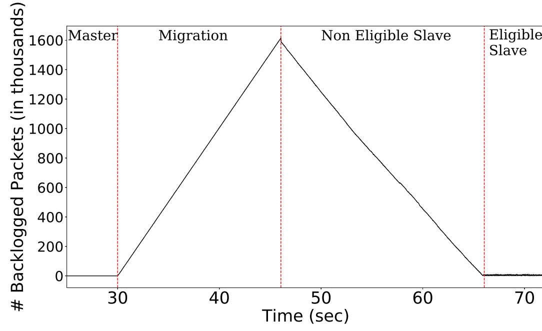
Specifically, a VNF goes through the states as described in the state diagram in Figure 10: as soon as new master VNF is chosen, the previous master (or any other eligible slave VNF) can start migrating. When the migration is completed, replay commands are issued to all *backlogged packets* (namely, all packets accumulated in the packet logger and not replayed to the VNF during the migration state). During the replay period, the VNF is a *non-eligible slave* as it significantly lags behind the master VNF, implying a prohibitively long switch-over time. When the number of backlogged packets (including all packets arriving during the replay process) first hits zero, the VNF becomes an *eligible slave*, implying that it can be chosen as a master. Note that as new packets arrive all the time, the number of backlogged packets fluctuates. The number of backlogged packets is typically small (and corresponds to the switch-over time) when the VNF is in eligible slave state, grows linearly with time when the VNF is in migrating state (assuming a constant packet rate), and decreases rapidly when the VNF is in a non-eligible slave state. See Figure 11.

A. Evaluation

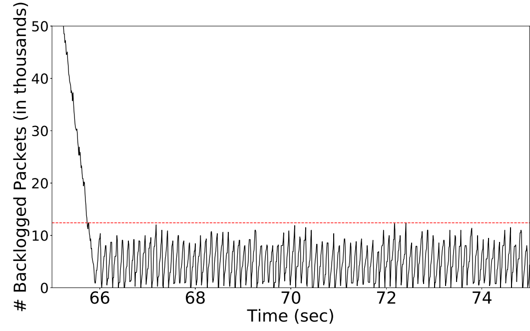
We have used FastClick [34] to implement most of our components (VNFs and the packet logger) and gRPC Remote Procedure Calls [42] for their communication. In addition, each VNF initiates a TCP session to the manager, whose main role is to exchange determinants (depending on the VNF role). Similarly, the manager holds a TCP session to the packet logger for replay and cleanup requests[4].

We ran our experiments using four VMware ESXi 6.5.0 hypervisors, that run on Dell Inc. PowerEdge R630 servers with 28 Intel(R) Xeon(R) CPU E5-2660 v4 2.00GHz processors. The four hypervisors are connected in a star topology via

[4]Our implementation is available at `github.com/reuvenperetz/-vnfmtd`



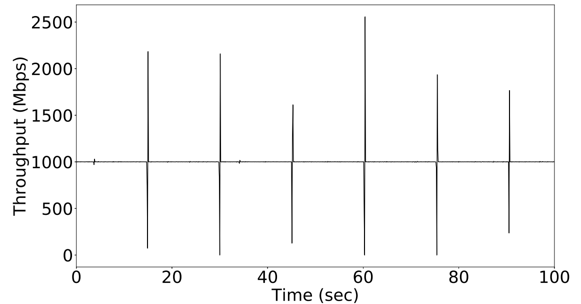
(a) The VNF goes through master state (0-30 sec), migrating state (30-46.05 sec), non-eligible slave state (46.05-66.04 sec), and eligible slave state (after 66.04 sec)



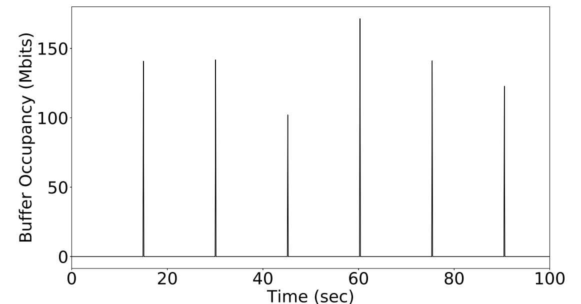
(b) The VNF is transitting from non-eligible slave to eligible slave. In this experiment, when working with 1Gbps traffic, during eligible slave the maximal replay traffic the packet logger needs to replay for that VNF is 124.11 Mbits, and the average is 49.15 Mbits.

Fig. 11. The number of backlogged packets in different VNF states.

Mellanox SN2100 LAB switches running SwitchDev. All VMs in our experiment use Ubuntu 16.04, 16 cores, 16GB RAM, and hard disk capacity of 60GB. We have run measurements at 1 Gbps with four threads dedicated to processing using DPDK to bypass the kernel. All our evaluations were made using three VNFs (one master and two slaves), but this is configurable and can be easily extended to a larger number of slaves. One of the hypervisors is used for running three virtual machines: the packet logger, the manager and a VM for traffic generation. The other three hypervisors run our three VNFs (each VNF on a different hypervisor). Figure 12(a) illustrates the throughput of the master VNF over time, as the manager initiates a switch-over process every fifteen seconds. Upon such a switch-over the master throughput transiently drops, as its buffer accumulates packets (see Figure 12(b)); when the switch-over is completed, packets are processed from the buffer (and not from the NIC, and therefore, there is a peak in throughput immediately after the switch-over). The entire switch-over process takes about 150 msec, which is several order of magnitude smaller than full-fledged VM migration (the exact number depends on the exact VM, in our setting VM migrations takes 14.8 seconds on average). Table I shows the average, maximum, and minimum times it takes until a *switch-over* is completed (namely, the time since the current master VNF stops processing packets until the new master starts processing packets, initially, from its buffer); *buffer draining time* is the time it takes the new master VNF to drain all buffers. After all buffers are drained, the master VNF behaves as if no switch-over occurred at all. If a full-fledged VM migration is required, the VNF is ineligible for around 33 seconds on average (which includes VM migration and processing backlogged traffic).



(a) The throughput of the master VNF over time.



(b) (Maximal) buffer occupancy over time.

Fig. 12. The throughput and buffer occupancy of the master VNF over time. Upon a switch-over, which occurs in this experiment every 15 seconds, the throughput transiently decreases and arriving packets are stored in the buffers. As the switch-over process completes, throughput is quickly recovered and the buffers become empty.

We have also studied the sensitivity of packet latency to the switch-over frequency and the number of determinants per packet.

Figure 13 shows the cumulative distribution function (CDF) of the latency and illustrates the influence of a switch-over frequency over our system. Naturally, the more frequent switch-overs are, the larger the latency is. However, there is no significant increase in latency when the switch-over frequency is set between 5-45 seconds.

Figure 14 shows that latency is more sensitive to the number of determinants generated for each packet, as this corresponding to changing shared variables and communicating with different components in our framework. We note that

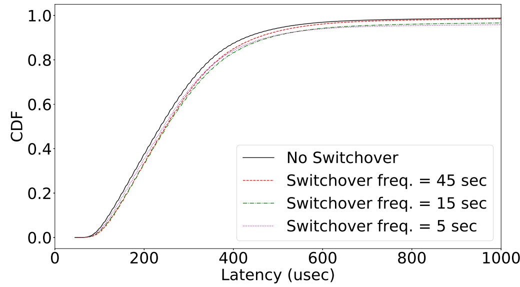


Fig. 13. CDF of packets latency for different switch-over frequencies.

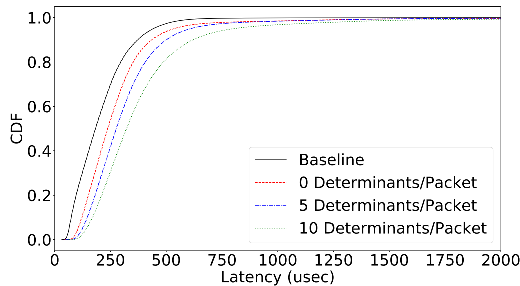


Fig. 14. CDF of packets latency for different number of generated determinants.

similar results were also reported in [10], [11], which also showed that the average number of determinants per packet is usually small. In our experiments, the throughput was not negatively affected when the master VNF generated up to twenty determinants per packet. However, when defending rare VNFs that generate significantly more determinants per packet (e.g., BW Monitor with 251 shared variables [10]), it might be better to use VM migrations only.

V. Conclusion and Future Work

In this paper, we demonstrated how common attack vectors such as DoS and side-channel could be mitigated using MTD approaches. We have introduced two novel techniques aimed at securing network functions by imposing an additional layer of uncertainty on the potential attacker, making the exploration and identification of weaknesses in the underlying system much harder, thus forcing an attacker to leave a heavier footprint. Specifically, we have proposed an MTD against algorithmic complexity attacks by proactively and reactively switching over modular blocks with equivalent function logic yet different implementation and behavior in an online running data-plane application. We have established that such an approach effectively counteracts specifically crafted adversarial workloads targeted at each specific implementation. We have also proposed an MTD approach for side-channel attacks due to co-residency. Our mechanism shuffles virtual network functions without inducing inefficient, heavy VM migrations. Thus, our framework reduces the amount of time each VNF might reside with a hypervisor hosting a malicious VNF.

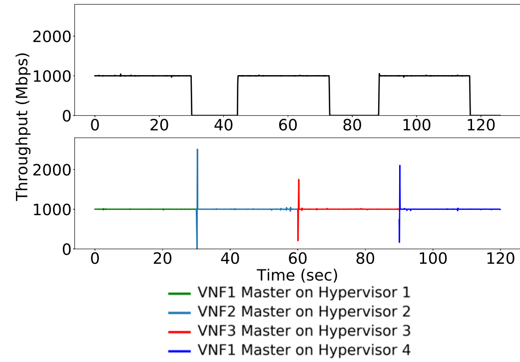


Fig. 15. The throughput of the VNF during VM migration with and without state migration. The top plot shows the throughput of a VNF when running VM migration every 30 seconds, while the bottom plot shows the observed throughput when combining VM migration with state migration, as illustrated in Figure 2.

Our paper can be extended in various directions to cover more attack vectors on VNFs or to improve the presented MTD techniques, e.g., by running multiple implementations of the same logic, some of which in different locations, and then measuring their performance can help to detect compromised VNF (which is currently an important challenge [1]).

In addition, combining both MTD approaches can help mitigate additional attack vectors. For example, to mitigate attacks that depends on the hypervisor type (see [1] for further details), one can have several implementations of the VNF, each running on a different hypervisor type, and then move the active (namely, master) VNF from one type of hypervisor to another.

Our MTD technique for algorithmic complexity attack can be further improved to combine ideas from MCA^2 [31], in which different implementation of the same block are run in parallel to provide resilience to attack with minimal throughput reduction in peace-time. The number of instances for each implementation keeps changing in response to the traffic.

Finally, as discussed in [43], an attacker might use a known format of communication passing through the network, as well as a known network application location to exfiltrate data or create covert channels inside the network. Using a proactive application version castling together with a proactive migration, the attack surface of the attacker is greatly reduced. Furthermore, a key technique to mitigate such an attack is to use *wardens*; namely, special network functions that capture communication in several locations of the network and analyze differences in the captured data. As wardens are VNFs, adding warden migration as discussed in Section IV may significantly increase the complexity of implementing these covert channels.

Acknowledgments: This work was partly supported by the Federmann Cyber Security Resarch Center in conjunction with the Israel National Cyber Directorate.

References

[1] W. Yang and C. Fung, "A survey on security in network functions virtualization," in *2016 IEEE NetSoft*, 2016, pp. 15–19.

[2] A. Bremler-Barr, Y. Harchol, and D. Hay, "Space-time tradeoffs in software-based deep packet inspection," in *IEEE HPSR*, 2011, pp. 1–8.

[3] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki, "Automated synthesis of adversarial workloads for network functions," in *ACM SIGCOMM*, 2018, pp. 372–385.

[4] S.-J. Moon, V. Sekar, and M. K. Reiter, "Nomad: Mitigating arbitrary cloud side channels via provider-assisted migration," in *ACM CCS*, 2015, pp. 1595–1606.

[5] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *ACM CCS*, 2009, pp. 199–212.

[6] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *ACM CCS*, 2012, pp. 305–316.

[7] ——, "Cross-tenant side-channel attacks in PaaS clouds," in *ACM CCS*, 2014, pp. 990–1003.

[8] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest os," in *ACM EUROSEC*, 2011, pp. 1:1–1:6.

[9] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: a software-defined framework for developing, deploying, and managing network functions," in *ACM SIGCOMM*, 2016, pp. 511–524.

[10] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker, "Rollback-recovery for middleboxes," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, 2015, pp. 227–240.

[11] Y. Harchol, D. Hay, and T. Orenstein, "FTvNF: Fault tolerant virtual network functions," in *ACM/IEEE ANCS*, 2018, pp. 141–147.

[12] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *USENIX Security*, 2017.

[13] M. Dunlop, S. Groat, W. Urbanski, R. Marchany, and J. Tront, "Mt6d: A moving target ipv6 defense," in *IEEE MILCOM*, 2011, pp. 1321–1326.

[14] J. R. Douceur, "The sybil attack," in *International workshop on peer-to-peer systems*. Springer, 2002, pp. 251–260.

[15] M. Albanese, A. De Benedictis, S. Jajodia, and K. Sun, "A moving target defense mechanism for manets based on identity virtualization," in *IEEE CNS*, 2013, pp. 278–286.

[16] D. C. MacFarland and C. A. Shue, "The SDN shuffle: creating a moving-target defense using host-based software-defined networking," in *ACM Workshop on Moving Target Defense*, 2015, pp. 37–41.

[17] Z. Zhao, D. Gong, B. Lu, F. Liu, and C. Zhang, "SDN-based double hopping communication against sniffer attack," *Mathematical Problems in Engineering*, 2016.

[18] Y. Zhou, W. Ni, K. Zheng, R. P. Liu, and Y. Yang, "Scalable node-centric route mutation for defense of large-scale software-defined networks," *Security and Communication Networks*, 2017.

[19] Y. Huang and A. Sood, "Self-cleansing systems for intrusion containment," in *ACM SHAMAN*, 2002.

[20] Y. Huang, D. Arsenault, and A. Sood, "Incorruptible system self-cleansing for intrusion tolerance," in *IEEE IPCCC*, 2006, pp. 493–496.

[21] J. H. Jafarian, A. Niakanlahiji, E. Al-Shaer, and Q. Duan, "Multi-dimensional host identity anonymization for defeating skilled attackers," in *ACM Workshop on Moving Target Defense*, 2016, pp. 47–58.

[22] Y. Wang, Q. Chen, J. Yi, and J. Guo, "U-TRI: Unlinkability through random identifier for sdn network," in *ACM Workshop on Moving Target Defense*, 2017, pp. 3–15.

[23] N. O. Ahmed and B. Bhargava, "Mayflies: A moving target defense framework for distributed systems," in *ACM Workshop on Moving Target Defense*, 2016, pp. 59–64.

[24] Z. Hu, M. Zhu, and P. Liu, "Online algorithms for adaptive cyber defense on bayesian attack graphs," in *ACM Workshop on Moving Target Defense*, 2017, pp. 99–109.

[25] S. Venkatesan, M. Albanese, G. Cybenko, and S. Jajodia, "A moving target defense approach to disrupting stealthy botnets," in *ACM Workshop on Moving Target Defense*, 2016, pp. 37–46.

[26] R. Skowyra, K. Bauer, V. Dedhia, and H. Okhravi, "Have no phear: Networks without identifiers," in *ACM Workshop on Moving Target Defense*, 2016, pp. 3–14.

[27] A. Chowdhary, S. Pisharody, and D. Huang, "Sdn based scalable mtd solution in cloud network," in *ACM Workshop on Moving Target Defense*, 2016, pp. 27–36.

[28] G. Dsouza, S. Hariri, Y. Al-Nashif, and G. Rodriguez, "Resilient dynamic data driven application systems (rDDDAS)," *Procedia Computer Science*, vol. 18, pp. 1929–1938, 2013.

[29] G. Rodríguez, M. J. Martín, P. González, J. Tourino, and R. Doallo, "CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 749–766, 2010.

[30] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks." in *USENIX Security*, 2003, pp. 29–44.

[31] Y. Afek, A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral, "MCA^2: multi-core architecture for mitigating complexity attacks," in *ACM/IEEE ANCS*, 2012, pp. 235–246.

[32] Y. Afek, A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral, "Making dpi engines resilient to algorithmic complexity attacks," *IEEE/ACM Transactions on Networking*, vol. 24, no. 6, pp. 3262–3275, December 2016.

[33] J. Steinberger, B. Kuhnert, C. Dietz, L. Ball, A. Sperotto, H. Baier, A. Pras, and G. Dreo, "DDoS Defense using MTD and SDN," in *IEEE/IFIP NOMS*, 2018, pp. 1–9.

[34] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *ACM/IEEE (ANCS)*, 2015, pp. 5–16.

[35] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[36] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *USENIX Security*, 2012, pp. 101–112.

[37] The Linux Foundation, "Data plane development kit (DPDK)," 2015. [Online]. Available: http://www.dpdk.org

[38] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[39] S. Wu and U. Manber, "Fast text searching allowing errors," *Communications of the ACM*, vol. 35, no. 10, pp. 83–92, 1992.

[40] "Clam AntiVirus," 2019. [Online]. Available: http://www.clamav.net(version0.101.4)

[41] E. Hjelmvik, "Hands-on network forensics," in *FIRST Conference*, vol. 11, 2015. [Online]. Available: https://download.netresec.com/pcap/FIRST-2015/FIRST-2015_Hands-on_Network_Forensics_PCAP.zip

[42] "gRPC," 2019. [Online]. Available: https://grpc.io

[43] W. Mazurczyk, P. Szary, S. Wendzel, and L. Caviglione, "Towards reversible storage network covert channels," in *ACM ARES*, 2019, p. 69.