FTvNF: Fault Tolerant Virtual Network Functions

Yotam Harchol
UC Berkeley
Berkeley, CA, USA
yotamhc@berkeley.edu

David Hay
The Hebrew University
Jerusalem, Israel
dhay@cs.huji.ac.il

Tal Orenstein
The Hebrew University
Jerusalem, Israel
tal.orenstein@mail.huji.ac.il

ABSTRACT

One of the major concerns about Network Function Virtualization (NFV) is the reduced stability of virtual network functions (VNFs), compared to dedicated hardware appliances. Stateful VNFs make recovery a complex process, where a major concern is how to handle non-determinism such as multi-threaded processing, time dependence, and randomness.

In this paper we present FTvNF — a new approach for network functions recovery with very low overhead in failure-free time. This is in contrast to previous suggestions to take snapshots of the VNF state at certain checkpoints or to store the VNF state externally. Compared with state-of-the-art approaches, our approach significantly reduces the latency overhead incurred by the network elements, both in failure-free operations and when failures occur. In addition, our approach better suits the common case of NFV service chaining, as our mechanisms are applied once per chain, thus significantly improve the performance over approaches that treat each VNF separately.

CCS CONCEPTS

• **Networks → Middle boxes / network appliances**; **Network reliability**; *Network services*; *In-network processing*;

KEYWORDS

Network Function Virtualization, NFV, Fault Tolerance, Service Chaining

ACM Reference Format:
Yotam Harchol, David Hay, and Tal Orenstein. 2018. FTvNF: Fault Tolerant Virtual Network Functions. In *ANCS '18: Symposium on Architectures for Networking and Communications Systems, July 23–24, 2018, Ithaca, NY, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3230718.3230731

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ANCS '18, July 23–24, 2018, Ithaca, NY, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5902-3/18/07…$15.00
https://doi.org/10.1145/3230718.3230731

1 INTRODUCTION

Network Function Virtualization (NFV) has become increasingly popular in large scale networks [6], where software virtual network functions (VNFs) are either replacing traditional hardware middleboxes or are added to the network beside existing middleboxes to provide flexible packet processing at a lower cost. While most hardware middleboxes provide hardware-based fault tolerance capabilities (e.g., redundant power supply, cooling, etc.), it becomes much more complex to provide such capabilities for software-based virtual appliances, that often run on commodity servers [19].

In this paper, we propose a novel framework for VNF fault tolerance, called FTvNF. It provides VNF state tracking, with a focus on lowering the overheads on failure-free executions. We take into account the fact that, nowadays, packets flow through multiple network functions before reaching their destination (a.k.a a *"service chain"*). Therefore, our proposed framework is designed to amortize the state tracking costs for multiple VNFs.

Recently, several works proposed frameworks for systematically providing fault tolerance capabilities for virtual network functions [21], where the solution is provided as a framework rather than specifically for each function. However, these solutions introduce non-trivial overhead on network function operation *even in cases where there was no failure at all.*

On the other hand, FTvNF is designed to incur minimal overhead in a failure-free operations. FTvNF deploys two instances of the protected VNF, in a master and a slave modes. In case of a failure, traffic is handled by the slave machine while the master is recovering.

Specifically, packets arriving at a service chain first going through an *ID sequencer* component that generates a unique ID for each packet and adds it to the packet. From the sequencer, packets are sent through the master VNFs (denoted M1,M2,M3 in Figure 1). All the packets are stored in a centralized logger, with persistent storage and capabilities to overcome from failures, until we are sure they were fully processed. A *manager* process observes the packet progress along the service chain (as recorded in another logger, called *determinant and progress (D&P) logger*) in an

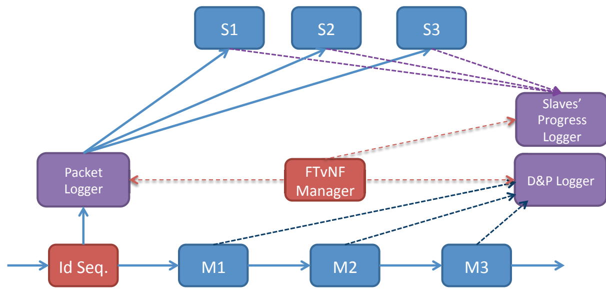

Figure 1: Overview of the FTvNF Framework

asynchronous manner; when some master VNF M_i is done with some packet p, the manager instructs the packet logger to send p to the corresponding slave S_i; packet p is deleted from the packet logger when the last master VNF *and the last slave VNF* in the chain finished processing it.

One of the major challenges of successfully providing fault-tolerance to network functions is to handle nondeterministic state changes correctly, such as time-based variables and random processes. FTvNF adapts the approach presented in [21] and carefully logs packet access data, 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 D&P logger. On the other hand, the slave VNF uses the values recorded in that logger (instead of performing the non-deterministic operation), and therefore, can exactly emulate the master VNF's behavior.

FTvNF deals with various failure scenarios, including failure of one or more master VNFs, failure of one or more slave VNFs, failure of a master VNF and its corresponding slave VNF, failure of the ID sequencer, and failure of the FTvNF manager process. All our loggers are, on the other hand, assumed to be fault-tolerant and are implemented using persistent storage (e.g., as in [1]).

The most important advantage of using our approach is that FTvNF does not use VNF snapshots at all. This is in contrast to previous works like [4, 21], where the entire system state should have been recorded at certain checkpoints, interfering with the system normal failure-free operations. As a result, FTvNF does not impose spikes of super-high latency as can be seen in other works that use such snapshots.

We have implemented the FTvNF framework and provided a preliminary evaluation of its performance on a small testbed with several physical servers running multiple VMs. Our preliminary experiments show that FTvNF incurs a marginal latency overhead of about 19% on a single VNF service chain, for failure-free times (with additional per-packet latency of 30 μs including two additional hops). This overhead is comparable to reported overhead in other state-of-the-art works. We show that this overhead is rapidly reduced for longer chains. We also show that in case of a failure, FTvNF is able to recover in 16.24 milliseconds, which is comparable (and even better) than previous works. Unlike previous works, our system keeps its normal operations while recovering from a failure (as traffic is diverted to the slave copy of the VNF). Furthermore, the amount of data needed to be re-processed upon recovery depends on the lag between the slave and the master, which is more flexible (and introduces a little effect on the system performance) than the frequency of snapshots in previous works.

2 RELATED WORK

Frameworks for fault tolerant network functions can be classified into three categories.

The first category, which we call *no-replay methods*, buffers the incoming and outgoing packets, and prevents the outgoing packets from being sent until the state is stored in a persistent storage. Works in this class include Colo [4] and Remus [3], which manage synchronization between VM replicas, and Pico [20], which provide per-flow synchronization using flow-level checkpoints. It is important to notice that all these frameworks suffer from periodical throughput degradation (namely, during snapshot operation).

The second category is *replay-based methods*. Frameworks in this category are based on replaying data between active master machine and a slave replication (e.g., [5, 15, 17]). In such systems, an application is being run in parallel on both the master and the slave, while changes done by the master are replayed to the slave to keep them synchronized. Using replay based synchronization is problematic for many VNF applications since they may use nondeterministic operations. Thus, these approaches cannot be directly applied on VNFs.

The current state-of-the-art work in this class is FTMB [21], which stores (for limited time) the packets of the traffic, as well as additional information (also called 'determinants'), and upon a failure reintroduce these packets to the network function in order to recover its state, using the determinants associated with each packet. In order to reduce the required persistent storage size and the number of packets that should be replayed upon a failure, FTMB takes periodical snapshots and stores it to persistent storage. Our approach falls under this category.

Finally, the third category aims to eliminate states in network functions (thus, greatly simplifying a recovery from failures) by *externalizing the network function state to a centralized layer*. While this simplifies the operations from a fault tolerance perspective, the main problem of these approaches is that they incur very high performance penalties on the network, even in failure-free times. It is important to note that the original motivation behind these approaches is

not always dealing with fault tolerance, but rather adapting common operations in virtual machine to an NFV setting (e.g., facilitating VNF migration). Works in this class include OpenNF [8] and StatelessNF [10]. OpenNF [8] suggests a centralized state storage and management, with the focus on state migration, while StatelessNF [10] suggests to decouple the state of a network function from its processing, allowing for each of them to perform its specific operations.

Finally, we note that as FTvNF uses two instances of the same VNF, there is an orthogonal question of where to place each VNF. This placement problem is discussed (in a different setting) in [11, 12].

3 FTVNF SYSTEM DESIGN

FTvNF deploys each VNF it protects twice, in a master and in a slave VMs. It takes a replay-based approach: packets are saved in a persistent logger when entering the service chain and are processed by the master VNFs. Asynchronously, and upon processing completion by the master VNF, packets are sent to the slave VNF. The slave VNF emulates the master VNF action by extracting the results of non-deterministic operation from a dedicated logger.

Our framework assumes that all the packets associated to the same flow are processed by the same thread and failures correspond to the *fail-stop failure model*. In that model a failed component ceases its operations immediately after it crashes and all neighbor entities are able to detect the failure (whether by detecting the link failure or by detecting the entity state) and stop forwarding new packets to it.

Our framework consists of the following main components.

ID Sequencer This chain gateway entity is responsible to generate a new ID and add it to the packet header. Several methods to add an ID to packet header exists in literature (e.g., [7, 9]) and can be used as is. In our implementation, however, we chose to push a 36 bit packet ID as three VLAN tags, each contributing 12 bits (namely, the VLAN identifier part only). Specifically, the 3 most-significant bits are used for the sequencer's own unique ID, the next 28 bits are used for sequence number that is generated by this ID sequencer, and the 5 least-significant bits are used for packet version (namely, to cope with situation in which the packet is changed along the chain). Notice that VLAN identifiers 0 and 4095 are reserved, so before assigning a new ID to a packet, we verify that all 3 VLANs are not assigned a reserved value. Moreover, the sequencer's own ID is part of the packet ID, so that different sequencers will produce different ranges of IDs. This is especially important for situations where a sequencer fails and a new sequencer should be launched instead.

After adding the ID to the packet, the sequencer sends, using an already-established TCP connection, the wrapped packet to the *packet logger*. Only after receiving an explicit acknowledgment from the logger (in the application layer) that the packet was stored, we can forward the packet to the first VNF of the chain.

Packet Logger The packet logger is in charge of storing and deleting packets, marked with their corresponding IDs, and their versions. Any VNF that changes a packet (either by rewriting its header or changing its payload) should increase the packet version and send the updated packet with the new ID to this logger before continuing to forward the packet along the chain. In order to shorten the time it takes to send a packet and to reduce the logger memory footprint, we save only the *difference* between the versions (e.g., if the TTL header is the only thing changed in a packet, there is not need to entire unchanged payload). Constructing a packet at some version v is done by fetching the entire base packet (namely, the packet with version 0), and applying the deltas saved in version $1, \ldots, v$. As mentioned previously, we assume that this logger (as all other loggers in the framework) is persistent and can overcome failures.

VNFs in master-mode During their course of operations, VNFs may access non-deterministic values. Some common examples are state variables. The values of such variables may be shared across packets, be determined by the specific packet processing order, and, in case of multi-threaded VNF, the thread scheduling within the VNF. Other examples include hardware-related outputs (such as the current time and random values). We call such values collectively *packet access data* (PAD) [1] and we distinguish between two kinds of such PADS:

Getter PADs (GPADs) are generated each time a packet reads a shared variable value or call a function that uses hardware or non-deterministic operations. It consists of a unique ID of the variable, the read value, and a per-variable sequence number of the read operation (which allows the process to recover to the last seen value).[2] Complementarity, *Setter PADs* (SPADs) are generated each time a VNF updates a value non-deterministically. SPADs are similar to GPADs except that the updated value is not stored, as it can be recalculated by the order of operations (and values stored in GPADs).[3]

At the end of the packet processing, the VNF wraps all generated PADs, adds metadata (packet ID and VNF ID), and

[1] In [21], such data is refereed to as PALs. [21] does not distinguish between PALs relating to getters and setter methods.

[2] For complex variables such as arrays and lists, we also store the *index* (e.g., of the element in the array that was read). Notice that if the VNF iterates over such an array, each element it accesses generates a separate GPAD.

[3] Furthermore, similar to GPADS, they have per-variable *update sequence number* field, which enables the logger to serialize updates to the same variables. For lists, SPADs also have an *operation* field, capturing list operations such as removal or insertion of elements.

sends them all to the D&P logger. If the packet has changed, it increases the packet version and sends the differences to the packet logger. It then waits for the loggers to acknowledge that these records were received. Only then, the VNF forwards the packet to the next VNF in the chain. Note that if no PADs were generated, the VNF can forward the packet immediately and should not communicate with the D&P logger at all.

We note that in addition to the PADs themselves, we store for each non-deterministic variable also the last seen value of that variable. This is required to facilitate our clean-up operation (in which packets and their corresponding PADs are removed from the loggers) and to ease our recovery operation (in this way, we can distinguish between situations were packet is processed more than once, and ignore such redundant processing).

VNFs in slave-mode When working in slave mode, VNFs use the PADs' records stored in the D&P logger by the corresponding master VNF. The slave VNF executes its code normally, but when accessing a non-deterministic value, it either takes the value from the corresponding GPAD (instead of generating the value from scratch) or makes sure that the set operations are executed in the same order as in the master VNF (namely, by looking at the corresponding SPAD and check its sequence number)

VNFs in slave mode do not produce PADs and should not forward packets. However, they need to record which packets they have successfully processed. This is done by sending the packet ID to a dedicated *Slaves' Progress Logger*.

Determinant and Progress (D&P) Logger The D&P logger is responsible for recording the traversal of packets through the service chain and the non-deterministic operations it went through (namely, by storing their PADs). As PADs are sent to the D&P logger only upon completion of the packet processing by the VNF, it implicitly records the progress of packets along the service chain.[4] This property becomes important when recovering VNFs that have failed.

Slave VNFs consume PADs from the D&P Logger: as they process some packet, they retrieve all PADs associated with that packet that were generated by the corresponding master VNF.

Slaves' Progress Logger This logger is responsible for recording which packets were processed successfully by each slave VNF. This progress recording becomes important when the master VNF fails and corresponding slave VNF should quickly take its place. Thus, the framework should know exactly which pending packets should be processed by the slave, immediately.

[4] It is important to notice that VNFs that do not store any PADs for any given packet are in fact stateless VNFs. Therefore, the correctness of our recovery scheme holds even if packets traverses these VNFs more than once. Hence, packets' progress over stateless VNFs should not be recorded.

FTvNF Manager The FTvNF manager has two major tasks. First, it is responsible of initiating packet processing in the slaves, which is done by signaling the packet logger to send specific packets to specific slave VNFs. In addition, it is responsible to delete packets that were processed by all (stateful) masters and slaves; packets that were intentionally dropped along the chain (e.g., by a firewall) are also taken into account by recording the drop action as a version change.

Specifically, the FTvNF manager periodically fetches the list of packets/PADs stored in the two loggers.

For each VNF i and packet x that was processed by M_i, the master copy of VNF i, and not by its slave copy, the FTvNF manager signals the packet logger to send the packet to the slave. In case there are many versions of that packet in M_i, only the first (earliest) version should be sent to the slave. As the slave records the processing of packet x as soon as it completes its corresponding operations, FTvNF manager needs to keep track of packets "in the air", so that packets are not processes twice by the slave. Notice that replaying the same packet several times to the slave does not raise a concern regarding its internal state (we explicitly attach a sequence number to each update operation, and perform an update if and only if its sequence number is larger than the last seen one), but can degrade the performance of the framework.

When a packet is processed by all master and slave VNFs, FTvNF manager deletes the packet from the packet logger, the corresponding PADs from D&P logger, and the packet ID from the slave progress logger. However, as the packet can no longer be replayed in the system upon a component failure, FTvNF manager must store the last value of each non-deterministic variable in persistent storage, before deleting the packet.

4 COPING WITH FAILURE SCENARIOS

In this section we describe in details the actions needed to be taken upon a failure of a component in the system.

Recovery from a master VNF failure Suppose master VNF i has failed, and suppose that M_i are the corresponding PAD records stored in the D&P logger. As we consider fail-stop model, it is safe to assume that the FTvNF manager is notified immediately upon such failure (this can be achieved, for example, by sending "keep-alive" messages from the VNF to the manager). The goal of the FTvNF manager is to make the slave available as soon as possible. This is done by contacting the slaves' progress logger, to check which packets it has already processed, denoted by S_i. As in its normal operation, it instructs the packet logger to send all packets in

$M_i \setminus S_i$ to the slave, and fetches all corresponding PADS from the D&P logger. When the last packet in M_i is processed by the slave, it reaches the same state of the master and is thus ready to replace it. This can be done simply by rerouting packets through it instead of the failed master (e.g., using a traffic steering capabilities, or any other mechanism used to route packets along a service chain).

Lazily, another slave VNF is launched. Then, all non-deterministic variables are set with the last recorded value which was computed by as well as the highest sequence number. This will account for all packets that have traversed the chain in the past but were already deleted from the packet logger (namely, all packets until the first packet stored in M_i).

Finally, all packets of M_i are sent from the packet logger to the slave along with the corresponding PADs. In the end of this process, the new slave will be in the same state as the failed master was upon its failure (it might be behind the new master VNF as more packets are processed meanwhile, however this is normal and will be fixed by the FTvNF manager upon its next periodic invocation).

The following process presents an important tradeoff in our framework: the more frequent the periodic invocations of FTvNF are, the shorter is the recovery from failures (mainly because $M_i \setminus S_i$ becomes smaller). Notice that as the FTvNF periodic invocations are done in the background on different threads (and many times on different CPUs on different machines), there is little overhead in increasing their frequency and by that reducing the recovery time. This is in contrast to [21], in which such invocation can be translated to taking a snapshot of the entire system, and therefore, introducing super-high latency spike, thus significantly limiting the frequency of these invocations.

Recovery from a slave VNF failure When the slave crashes a new slave should be invoked. This is done in exactly the same manner as done for the new slave that is launched when a master VNF fails and its slave VNF takes its place.

Recovery from simultaneous master and slave VNFs failures It this case, we are invoking two slaves and proceeding in parallel with both of them in the same way described before. Then, one of the slaves is chosen arbitrarily to be the new master VNF, implying traffic is routed through it and it starts sending PADs to the D&P logger.

Recovery from ID sequencer failure While the ID sequencer is a crucial component in our framework, it contains very small state information (namely, the last sequence number given to a packet). In addition, we do not rely on the sequence number of packets to determine their order, and therefore, it is sufficient to make sure that no two packets in the framework are given the same ID. This is done by attaching a sequencer ID to the packet. In such a case, a new-launched sequencer will have a different ID, implying it will assign packet IDs from a different range.

Recovery from FTvNF manager failure As mentioned earlier, this is a stateless process and therefore a newly-launched FTvNF manager process can easily take its steps.

5 IMPLEMENTATION AND PRELIMINARY EVALUATION

We have implemented FTvNF including all components described in this paper.

We have used Click [13] to implement our ID sequencer, master NFs, and slave NFs. Click is a modular software architecture for creating a software router from fine-grained components, called also elements. Recent works show that Click can also be used to create (virtual) network function (e.g., OpenBox [2]).

Beside integrating the above Click elements in the Click network function elements chain (Click network functions defines directed graphs of elements), it is necessary to update other elements used by the VNF that access to shared resources so that they will generate proper PAD for each access to shared state variable or hardware data. The code changes can be done by adding user annotations to each shared resource and building a suitable LLVM compiler [16], exactly as was done in FTMB (see [21] for more details).

All loggers in the system use the same simple implementation of a TCP server that accepts connections from the corresponding components and stores the received information in a data storage. These loggers can be easily replaced with other, more sophisticated, remote storage frameworks.

We ran our experiments on three commodity servers with 4 cores, 32GB RAM and two NICs each. In order to measure the packets round trip time, we have connected each of the servers to the others two with 10Gbps cables, in a ring topology. This allows us to measure the results on the same server clock. We have used KVM [14] as our virtualization environment and Open vSwitch [18] for switching packets between servers and virtual machines. Specifically, our testbed consists of three OVS bridges and several virtual machines, running VNFs and our framework component. Notice that, as we have only three servers in our setting, some of the components in our preliminary experiments run on the same physical server (and therefore might yield better results). FTvNF itself is not constrained by the number of servers.

We have injected real TCP packets from Alexa website to our testbed to measure the performance of FTvNF, where we consider both failure-free operations and the recovery operation. The Alexa trace was made by crawling to the top 100 websites listed by Alexa and downloading their content, up to 6 levels deep.

	Overhead (%)	Overhead (microseconds)
1	12.67	30.58
2	11.78	43.41
3	8.74	53.15

Table 1: FTvNF overhead as a function of chain length for VNFs with a processing time of 95 microseconds

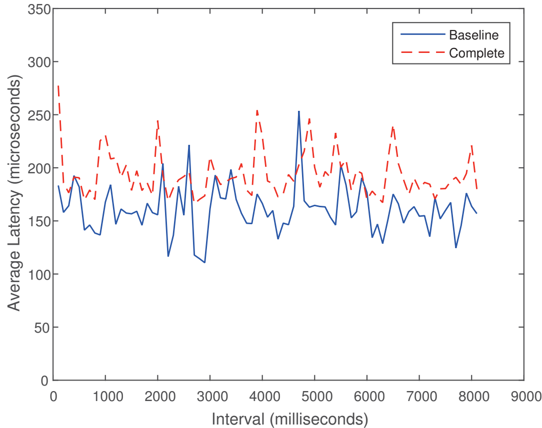

Figure 2: The latency of packets in the baseline and under fully-operational FTvNF over time.

Failure-free operations We distinguish between four sources of overhead that packets might incur during a failure–free operation: (i) going through an extra hop (namely, the ID sequencer); (ii) ID sequencer operations; (iii) changes in the (master) VNF; (iv) communication between the (master) VNF and the loggers.

We have tackled these overheads one by one. Naturally, adding more elements and operations increases the overall latency, but this increase is marginal. Even in a single VNF (namely, service chain of length 1), there is only 19% difference (which stands for 30 microseconds), between the median packet latency in the baseline and the median packet latency in a fully operational FTvNF. Most of the extra delay is due to the extra hop and ID sequencer (performed only once per chain). Thus, the relative penalty decreases as the service chain length increases. Table 1 depicts the relative overhead for our experiments with service chains of length two and three.

Figure 2 shows the packet latency measured over time, both in the baseline of a service chain of length one and in the fully-operational FTvNF settings. Evidently, packet latencies are stable and FTvNF does not introduce significant jitter to the system. This is in contrast to snapshot based scheme in which spikes of order(s) of magnitude larger latency are observed.

Recovery operations We have also checked the time it takes for our framework to recover from a master VNF failure. That is, the time it takes to send all necessary packets from the packet logger along with their PADs, sent from the D&P logger, and for the slave to process them correctly. The recovery time strongly depends on the number of packets that need to be replayed at the slave host. As mentioned before, this is bounded from above by the maximum number of packets that arrive at the system between two periodic operations of the FTvNF manager. Our recovery time, in most settings, is 4.5–8.0 milliseconds, where the difference in fail-free operation latency is negligible. It is important to notice that our recovery operation times are significantly less than existing frameworks (c.f, [21, Figure 11]).

Comparison with FTMB We have showed that FTvNF significantly reduces the recovery time and the latency overhead of failure-free operations over FTMB [21]. However, this improvement comes with an extra cost: While FTvNF doubles the computation resources as it uses two copies of each VNF, snapshot-based approaches are required to compensate for throughput penalty due to snapshotting. In FTMB, snapshotting is the primary cause of throughput penalty (rather than PAL insertions and other overhead) [21]. Specifically, a throughput reduction of 12.5% is experienced in Simple-NAT network function. Assuming that PAL insertion is negligible in that case, FTMB requires a factor of 1.143 more computation resources to achieve the same throughput.

In the decision of whether to use FTMB or a similar approach, versus FTvNF's approach, one should take into account the benefits and the implications of each approach: Although FTvNF requires more computation resources, FTMB requires more storage, has extra latency in failure-free operations, and has longer recovery time.

While this is a resources vs. latency tradeoff, network administrator can easily enjoy the best of both worlds by combining these approaches, on VNF-by-VNF basis.

6 CONCLUSION

This paper has introduced FTvNF, a novel framework to provide fault tolerant virtual network functions. FTvNF is based on running a slave copy of the virtual network function along side a master copy of that function. This approach provides better availability compared to other state-of-the-art solutions that use snapshots, and can be used simultaneously with such approaches in order to suit the exact performance needs of each VNF.

Acknowledgments: Part of this work has been supported by the HUJI Cyber Security Center in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office and by the Chief Scientist Office of the Israeli Ministry of Industry, Trade and Labor within the "Neptune" consortium.

REFERENCES

[1] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters.. In *NSDI*. 1–14.

[2] A. Bremler-Barr, Y. Harchol, and D. Hay. 2016. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *SIGCOMM*. 511–524.

[3] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*. 161–174.

[4] Y. Z. Dong, W. Ye, Y. H. Jiang, I. Pratt, S. Q. Ma, J. Li, and H. B. Guan. 2013. COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service. In *SOCC*. Article 3, 16 pages.

[5] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. 2008. Execution Replay of Multiprocessor Virtual Machines. In *VEE*. 121–130.

[6] ETSI. 2012. Network Functions Virtualization - Introductory White Paper. (2012). http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[7] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. 2014. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags. In *NSDI*. 533–546.

[8] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*. 163–174.

[9] J. Gross, T. Sridhar, P. Garg, C. Wright, I. Ganga, P. Agarwal, K. Duda, D. Dutt, and J. Hudson. 2015. Geneve: Generic Network Virtualization Encapsulation. IETF Internet-Draft. (November 2015). https://tools.ietf.org/html/draft-ietf-nvo3-geneve-00.

[10] M. Kablan, A. Alsudais, , E. Keller, and F. Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *NSDI*. 97–112.

[11] Y. Kanizo, O. Rottenstreich, I. Segall, and J. Yallouz. 2017. Optimizing Virtual Backup Allocation for Middleboxes. *ACM Transactions on Networking* 25, 5 (Oct. 2017), 2759–2772.

[12] Y. Kanizo, O. Rottenstreich, I. Segall, and J. Yallouz. 2018. Designing Optimal Middlebox Recovery Schemes with Performance Guarantees. In *INFOCOM*.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. 2000. The Click Modular Router. *ACM TOCS* 18, 3 (2000), 263–297.

[14] KVM. 2017. Kernel Virtual Machine. (2017). https://www.linux-kvm.org/page/Main_Page.

[15] O. Laadan, N. Viennot, and J. Nieh. 2010. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *SIGMETRICS*. 155–166.

[16] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*.

[17] J. R. Lorch, A. Baumann, L. Glendenning, D. Meyer, and A. Warfield. 2015. Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-Tolerant Services. In *NSDI*. 575–588.

[18] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. 2015. The Design and Implementation of Open vSwitch. In *NSDI*. 117–130.

[19] R. Potharaju and N. Jain. 2013. Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters. IMC.

[20] S. Rajagopalan, D. Williams, and H. Jamjoom. 2013. Pico Replication: A High Availability Framework for Middleboxes. In *SOCC*. 1:1–1:15.

[21] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. 2015. Rollback-recovery for middleboxes. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 227–240.