TopoFuzzer — A Network Topology Fuzzer for Moving Target Defense in the Telco Cloud

Wissem Soussi [12], Maria Christopoulou [3], Themis Anagnostopoulos [3], Gürkan Gür [2], Burkhard Stiller [1]
[1]Communication Systems Group CSG, Department of Informatics IfI,
University of Zürich UZH, Switzerland [soussi|stiller]@ifi.uzh.ch
[2]Institute of Applied Information Technology (InIT), Zurich University of Applied Sciences,
Switzerland [sous|gueu]@zhaw.ch
[3]National Center for Scientific Research Demokritos (NCSRD),
Greece [maria.christopoulou|thmanagnostopoulos]@iit.demokritos.gr

Abstract—**Telecommunication networks are shifting to multi-cloud environments. This trend is expected to shape the standardization and implementation of future networks. Thus, the protection of virtualized services has become more critical. One of the promising methods to secure virtual resources in that setting is Moving Target Defense (MTD). This paper presents the Network Topology Fuzzer (TopoFuzzer) module, enabling different MTD operations that change the topology of a 5G network. An emphasis is given to live re-instantiations and live migrations of running services and, consequently, security gains against Advanced Persistent Threats (APTs). This work utilizes a 5G testbed to evaluate the TopoFuzzer module and MTD operations on Virtual Network Functions (VNFs).**

Index Terms—**Moving Target Defense (MTD), live service movement, 5G Network Management and Orchestration.**

I. Introduction

In telecommunication networks, the management and orchestration of virtualized network resources is already a broad and fundamental research topic [1]. Following the Network Function Virtualization (NFV) architecture [2], telecommunication networks deploy network services (NSs) and virtual network functions (VNFs) in cloud infrastructures as a set of virtual machines (VMs) or container runtimes. This architectural paradigm is also expected to be a fundamental part of future networks like 6G.

In general, virtual resources can be orchestrated for security purposes using Moving Target Defense (MTD) principles [3]. MTD periodically alters a network's topology and respective configurations, changing the attack surface constantly to make the network a moving target for attackers. The objective is to reduce the attacker's effective action space in *time* and *space*. With the former, attackers have to perform the attack in a reduced amount of time to make it possibly successful or to exploit the collected intelligence. With the latter, the attackers are forced to re-scan the network and gather new data as previous reconnaissance becomes obsolete, *i.e.*, a feasibility and overhead problem for attackers. However, changing or moving a running telecommunication network might also disrupt its service and its availability, heavily impacting the Quality of Service (QoS) and possibly violating Service Level Agreement (SLA) requirements.

This work considers MTD operations, such as automated migrations and re-instantiations of VNFs, and discusses the benefits and possible concerns that may arise. To address both explicitly, this paper introduces "TopoFuzzer", a novel solution to reduce the overhead of MTD operations on QoS of services deployed and used by end users. Specifically, the contributions of the TopoFuzzer and its underlying work are summarized as follows: ***1)*** The design, architecture, and development of a network module for network topology fuzzing. ***2)*** Enabling moving stateless VNFs, while they are being used and without service disruption. ***3)*** Changing the network topology with limited resources and network overhead, utilizing Virtual Network Interface Cards (vNIC), also preserving 5G network slicing isolation ***4)*** Measurement-based data and analysis for empirical evaluation. ***5)*** An open source implementation, publicly available at GitHub [4].

II. Technical Background and Related Work

For implementing MTD in virtualized network environments, a major design choice, as well as a research question, is what actions are available and reasonable for security protection. In general, two main categories define the set of possible MTD actions:

- ***Soft MTD actions***: These involve the modification of the network topology, *e.g.*, using SDN-based shuffles of traffic flows, reconfiguring network interfaces, or replacing switches and routing nodes. To this scope, TopoFuzzer serves as a middle virtual network.
- ***Hard MTD actions***: These operations directly modify the services' VNFs, *e.g.*, migrating a VNF to a different location or re-instantiating it with an authenticated image.

The advantages of Hard MTD actions rely mainly on the usage of an authenticated image, removing from the service any post-instantiation infection and Advanced Persistent Threat (APT) originating from malware (*e.g.*, installed backdoors, spyware, botnet command and control (C&C), ransomware, or other hijacked services). Moreover, a service can increase the fault-tolerance to its host, by migrating the VNF to a different edge node, in case the current node is under attack or simply under maintenance.

In an NFV environment, these actions are typically enforced by a controller logic for executing the MTD operations on the VNFs by interacting with the NFV Orchestrator. As an example of this design, Soussi et al. have proposed an MTD Controller module, MOTDEC, to integrate with the new NFV Security Manager component defined in the ETSI NFV-SEC 024 draft standard in [5]. MOTDEC directly enforces soft MTD actions via the TopoFuzzer module presented in this work. Hard MTD actions, instead, additionally need the coordination of the NFV MANO and Network slice Manager.

A. Traffic Redirection

The traffic redirection to the new service instance has to be transparent, meaning that the shift to the new instance has to be unnoticed by users and possible attackers. Each VNF has a public IP address and a private one. During hard MTD actions, the public IP is immutable and fixed, while the private IP changes each time a new instance replaces an old one.

When dealing with UDP based communications, the traffic redirection is done by using a simple Network Address Translation (NAT) gateway. As UDP is connectionless at the transport layer, packets are accepted and directly pushed to the application layer. However, when dealing with TCP based traffic, each packet is bound to a connection defined by the tuple (`src_ip, src_port, dst_ip, dst_port`). Using simple NAT will work only for newly established connections, while running connections keep using the old private IP as the destination until their termination. Waiting for persistent connections to terminate might be unfeasible or highly costly in terms of resource consumption as two instances of the same service are forced to run simultaneously. Other NAT-based techniques, such as Hole Punching [6], have the same issue with TCP and may expose the private IP of the VNF when trying to establish a direct connection. Binder et al. [7] showed that the connection could be handed over to a new server by changing IP packet fields related to the "TCP state machine" such as SYN/ACK number, congestion window, and checksum value.

In the scope of redirecting traffic to a honeypot server, Cunha et al. [8] introduced a traffic redirection process using a proxy with two sockets: *i)* an `in_socket`, to establish a connection with the client, and *ii)* an `out_socket`, to establish a connection with the moving server. The client node can only see the connection established with the public IP of the VNF used by the proxy. When the VNF moves, a new connection is established using the `out_socket` and sending the payload to the new server. In [8], it is shown that this method allows redirecting the traffic with less impact on the performance compared to the SDN method used in [7].

While the use case defined in [8] only considers redirecting the traffic of a service running behind one specific port (*e.g.*, a web server on port 80), VNFs are full servers with possibly all the ports open. Moreover, a different proxy node is used for each VNF as its NIC is mapped to the VNF's public IP, interfacing the service to the client.

TopoFuzzer uses the two-socket proxy method inspired by [8] but excludes the usage of the `TCP_REPAIR` kernel module. It also solves the above-mentioned problems of redirecting traffic to only one port and uses one proxy for one VNF, maintaining slice isolation. Finally, in contrast to the aforementioned works, the implementation of TopoFuzzer is made open source to facilitate reproducibility and future research developments.

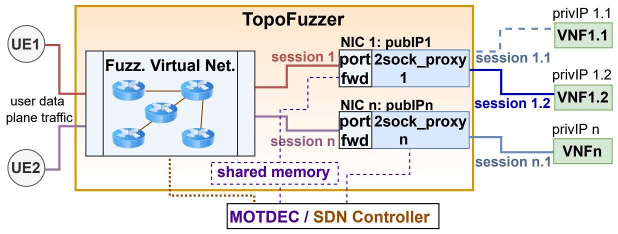

Fig. 1: TopoFuzzer architecture

III. TopoFuzzer Architecture

TopoFuzzer architecture is depicted in Figure 1, which comprises three main components: *1)* The shared memory containing an IP mapping table, *2)* The redirection module composed of multiple proxy-NICs, and *3)* The Fuzzing Virtual Network allowing to implement Soft MTD actions.

A. IP mapping and interface to MOTDEC

Public IPs are predefined for each VNF as they remain fixed during Hard MTD actions. When a VNF is deployed, the private IP assigned by the Virtual Infrastructure Manager (VIM) hosting the VNF is given to the MTD controller MOTDEC, which then sends both IPs to TopoFuzzer. TopoFuzzer maintains the mapping between private and public IPs. The mapping is a hash table with two entries per mapping (i.e., `pubIP`$\rightarrow$`privIP` and `privIP`$\rightarrow$`pubIP`). The hash table, implemented in Redis [9], is then accessible with an API interface for communication with MOTDEC.

B. 2-socket proxy redirection improvement using `conntrack` and port forwarding

A server proxy instance binds the `in_socket` to a fixed known port. A port forwarding rule is going to change the destination port of all the traffic sent to a VNF. This allows the proxy to receive all the traffic despite listening to one port. In order to get back the original destination port, TopoFuzzer uses a Linux Kernel feature that keeps track of all the connections and that is used to enable NAT operations, namely `conntrack` [10]. To identify the right connection of forwarded packets, TopoFuzzer uses three available values: the connection type (which, in this case, is always TCP), the source IP of the client, and the source port of the client. Knowing the connection is session based, the pair (`source_IP, source_port`) is unique per connection (contrary to UDP, where the same pair can send packets to different IPs and ports). The set of (vNIC, proxy) per VNF is implemented using the Mininet [11] host abstraction, which creates a new vNIC and starts separate python interpreters (for the 2-sockets proxy function) per host, keeping them in the same Linux kernel space of the TopoFuzzer node. This implementation proves to be considerably flexible and efficient as it uses the same amount of resources while quickly scaling up TopoFuzzer for large-scale realistic networks with hundreds of VNFs. Moreover, it reduces the network overhead of the TopoFuzzer solution since the proxy nodes get direct system access to the mapping table without using network resources for the data exchange. As Redis is an in-memory hash-table store, fetching a new private IP for a specific connection has a constant complexity of $\theta(1)$.

C. Fuzzing Virtual Network

To perform Soft MTD actions without affecting the real network topology but changing the topology view from the client perspective (e.g., if they scanned the network with tools like `traceroute` and `nmap`), a virtual network composed of switching and routing nodes is placed between the proxy nodes and the User Plane Function (UPF). Adding a gateway in the route, removing one, or replacing it with another gateway, are all operations that affect only the visible session to the clients. In contrast, the sessions from the out_sockets of the proxy nodes to the VNFs are internal by nature, hence, not modified. These operations are enforced by using an SDN controller connected to the switches of the Fuzzing Virtual Network. The SDN Controller is hosted in the MOTDEC module, as it orchestrates MTD operations, while the Mininet API is used to initiate the fuzzing virtual network and the proxy nodes at runtime.

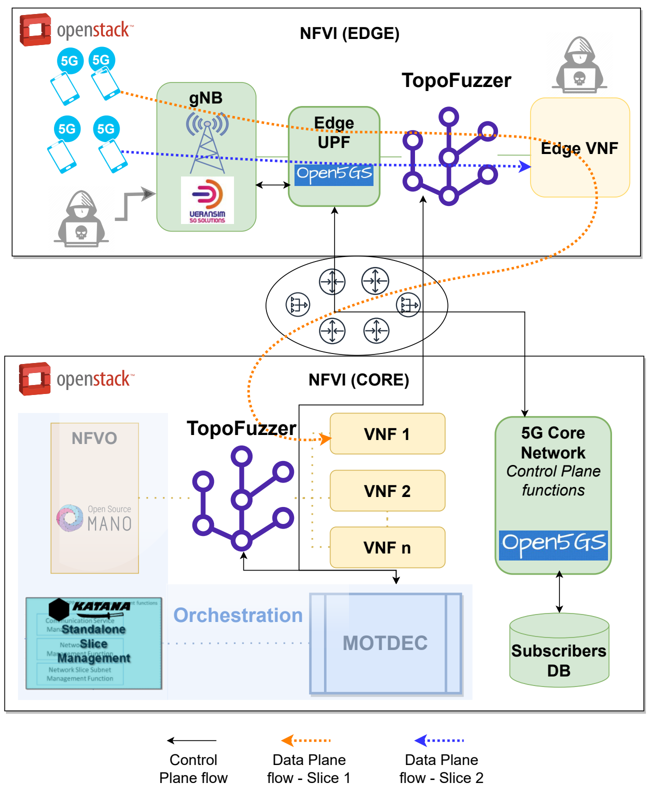

Fig. 2: 5G testbed deployment for TopoFuzzer

TABLE I: Average bandwidth for different traffic types

Traffic type	Avg. bandwidth Sender (Mbps)	Avg. bandwidth Receiver (Mbps)	Standard deviation
Direct TCP	159	159	8.95
Direct UDP	225	138	37.85
Indirect TCP	157	157	10.22
Indirect UDP	221	136	39.26

IV. 5G Testbed

Fig. 2 illustrates the topology of the TopoFuzzer 5G testbed, comprising two separate OpenStack deployments for the Edge, Radio Access, and Core network domains. The Edge Openstack NFVI includes the 5G UEs, the gNB, the Edge User Plane Function (UPF), and a generic VNF that provides an application service to the connected users. The Core Openstack includes the control plane of the 5G Core Network, the subscriber database, and generic VNFs for service provisioning. The Core Openstack also provides the MTD orchestration entities, namely the MOTDEC, the Katana Slice Manager [12], and the Network Function Virtualization Orchestrator (NFVO) implemented by Open-Source MANO (OSM) [13]. A Topofuzzer is deployed at both Edge and Core Openstack instances. This deployment allows emulating a distributed UPF architecture, where the UPFs are co-located with the gNB on the Edge domain. The control plane of the 5G network is deployed on the Core OpenStack and includes the control plane functions of the 5G network.

The core 5G network is implemented with Open5GS [14], an open-source 3GPP Release-16 compliant implementation. The Radio Access Network (RAN) and the mobile UEs are implemented by UERANSIM [15], an open-source UE and gNB simulator. The 5G architecture is Standalone (5G SA). UERANSIM connects to Open5GS via a control interface with the AMF and a user interface to the UPF. The UEs and the gNB connect via a simulated radio interface. Unlike actual hardware equipment, UERANSIM allows the deployment of multiple virtual UEs to test the solution's scalability, *i.e.*, the operational cost of the solution under an increasing network workload.

V. Experimental Results

To evaluate the TopoFuzzer overhead on connection bandwidth, both TCP and UDP traffic is generated from the UE using `Iperf3` [16]. As illustrated in Table I, TopoFuzzer did not show any relevant overhead on the bandwidth in the 5G testbed environment. However, what became evident during the tests is the importance of the allocated networking hardware resources and their capabilities. In our 5G testbed, the communication bottleneck was the UPF, which manages the data plane communication both at the core and edge domains (each domain with a different instance of the service).

A. TopoFuzzer QoS Overhead - HTTP/2 and HTTP/3

This set of tests runs HTTP/2 communication traffic to evaluate the QoS overhead based on latency and packet loss rate metrics. The RTT is used for determining the latency, while a request that does not receive a response within one second is counted as a packet loss. When establishing connections with 30 different UEs, in case of adding a UE every 5 seconds, TopoFuzzer overhead on the RTT is 4.5%, while for 50 UEs, this increases to 29% (from an average RTT of 23.384 ms to 30.35 ms). In both cases, no packet is lost.

To test the scalability of TopoFuzzer, 100 connected UEs are deployed by adding 5 UEs simultaneously every 5 seconds. The RTT overhead reaches 38% as depicted in Figure 3). RTT values with (orange line) and without TopoFuzzer (blue line) are the same most of the time (showed by the measured latency overlap depicted in Figure 3).

As TopoFuzzer can also redirect communications over the QUIC protocol (built over the UDP transport protocol), the QoS overhead is tested on the recently standardized HTTP/3 protocol. The first noticeable difference compared to HTTP/2 is the reduction of the latency overhead, with 4% when having 50 UEs connected (against the previous 29% in HTTP/2) and 0.9% when having 70 UEs (compared to 37% in HTTP/2). With and without TopoFuzzer, the testbed could not scale to 100 HTTP/3 UEs, assuming that the cause is the additional TLS encryption the VNFs have to support with their limited resources (1 vCPU and 512 MByte of RAM).

Another observable difference is the absence of the limitation on the number of simultaneous connection establishments: instead of seven simultaneous connections with HTTP/2, the number of connections with HTTP/3 has to be over 48 to start observing packet losses. However, this limit is the same when running direct HTTP/3 traffic (*i.e.*, without TopoFuzzer), inducing that it might have reached the limits of the resources allocated for the VNFs. Hence, the latency gap between direct and indirect communication does not increase.

Tests alternate between re-instantiate and migrate operations on the VNF 30 times using HTTP/2 and 30 times using HTTP/3. During a [re-instantiate – migrate – re-instantiate]

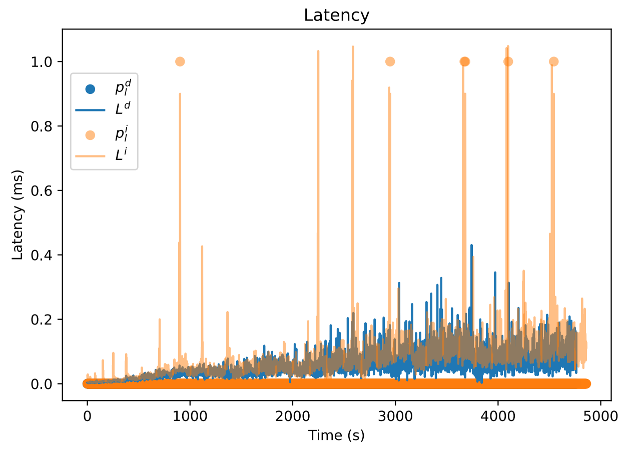

Fig. 3: HTTP/2 QoS overhead – Latency overhead and packet loss rate of TopoFuzzer (orange), compared to normal performances (blue), when increasing UEs by five every five seconds up to 100 UEs

sequence, the QoS overhead becomes observable with 10 UEs connected to the VNF via HTTP/2. The average packet loss rate increase in a one-second frame window is 7% for the VNF re-instantiations and 33% for VNF migrations. Packet losses affect the latency (*i.e.*, the RTT), occasionally up to hundreds of milliseconds for one second. Considering the measured data, where a 33% packet loss rate in one second corresponds to a downtime of around 330 ms, if a Service Level Agreement (SLA) with 99.999% of availability has to be maintained, TopoFuzzer would allow up to 911 MTD migrations or 4293 MTD re-instantiations per service per year. This is equivalent to 17 migrations or 82 re-instantiations per week, an upper limit sufficient to proactively neutralize and kick off any infection from the VNF.

When only one UE is connected to the VNF, there is no latency or packet loss rate increase when performing migrations and re-instantiations. With HTTP/3, the traffic load of 10 UEs shows a migration's latency overhead of 5 ms in a one second window, from 5 ms to 10 ms, before going back to 5 ms. Unlike HTTP/2, there is no packet loss during the simultaneous redirection of the UEs traffic, and this scales up to 50 simultaneous UEs. The downtime of 5 ms for HTTP/3 would allow performing up to 60126 migrations per year (and 283338 re-instantiations) under a 99.999% SLA availability requirement. To generally keep the values mentioned in terms of SLA requirements, TopoFuzzer could be optimized to progressively migrate connections based on its vNIC capability and the protocol involved (TCP or UDP) rather than simply redirecting all existing sessions at once. This comes with an additional cost stemming from running two instances of the service for a longer time, which might be a negligible cost compared to having an occasional service downtime.

VI. Summary and Preliminary Conclusions

TopoFuzzer enables the live migration of open connections built on TCP and UDP. HTTP/3 reduces the redirection complexity, improving its flexibility and QoS overhead compared to HTTP/2. TopoFuzzer shows a continuous bandwidth overhead of 1.33% for UDP and 1.25% for TCP, when services are not moving. QoS metrics, namely jitter and RTT, show a negligible increase when moving connections, with the best performance on UDP-based connections. For HTTP/3 a respective increase of 1.33% and 4% was achieved. Overall, this capability is instrumental and enables MTD proactive and reactive protection against any malware infection that leads to critical security threats such as APTs.

VII. Acknowledgement

The work described in this paper received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 871808 (5G-PPP project INSPIRE-5Gplus). The paper reflects only the authors' views. The Commission is not responsible for any use that may be made of the information it contains.

References

[1] P. Porambage, G. Gür, D. P. Moya Osorio, M. Livanage, and M. Ylianttila, “6G security challenges and potential solutions,” in *2021 Joint European Conference on Networks and Communications 6G Summit (EuCNC/6G Summit)*, 2021, pp. 622–627.

[2] “Network Function Virtualization (NFV),” https://www.etsi.org/technologies/nfv, European Telecommunications Standards Institute (ETSI), Nice, FR, Standard, Mar. 2012, [Online; accessed 21-October-2022].

[3] S. Sengupta, A. Chowdhary, A. Sabur, A. Alshamrani, D. Huang, and S. Kambhampati, “A survey of moving target defenses for network security,” *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 1909–1941, 2020.

[4] Wissem Soussi, “Topofuzzer - a network topology fuzzer.” [Online]. Available: https://github.com/wsoussi/TopoFuzzer

[5] W. Soussi, M. Christopoulou, G. Xilouris, and G. Gür, “Moving target defense as a proactive defense element for beyond 5G,” *IEEE Communications Standards Magazine*, vol. 5, no. 3, pp. 72–79, 2021.

[6] B. Ford, P. Srisuresh, and D. Kegel, “Peer-to-Peer communication across network address translators,” in *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. [Online]. Available: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/peer-peer-communication-across-network-address

[7] A. Binder, T. Boros, and I. Kotuliak, “A SDN based method of TCP connection handover,” in *Information and Communication Technology*, I. Khalil, E. Neuhold, A. M. Tjoa, L. D. Xu, and I. You, Eds. Cham: Springer International Publishing, 2015, pp. 13–19.

[8] V. A. Cunha, D. Corujo, J. P. Barraca, and R. L. Aguiar, “Using Linux TCP connection repair for mid-session endpoint handover: a security enhancement use-case,” in *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2020, pp. 174–180.

[9] Salvatore Sanfilippo, “Redis.” [Online]. Available: https://github.com/redis/redis

[10] Pablo Neira Ayuso, “Conntrack-Tools.” [Online]. Available: https://conntrack-tools.netfilter.org

[11] Mininet, “Mininet.” [Online]. Available: https://mininet.org

[12] Themis Anagnostopoulos, “Katana network slice manager.” [Online]. Available: https://github.com/medianetlab/katana-slice_manager

[13] ETSI, “Open source mano.” [Online]. Available: https://osm.etsi.org/

[14] Sukchan Lee, “Open5gs.” [Online]. Available: https://open5gs.org/

[15] Ali Güngör, “Ueransim.” [Online]. Available: https://github.com/aligungr/UERANSIM

[16] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, Kaustubh Prabhu, “Iperf3.” [Online]. Available: https://iperf.fr/