Cybersecurity

RESEARCH

Open Access

NDFuzz: a non-intrusive coverage-guided fuzzing framework for virtualized network devices

Yu Zhang[1,2,3,4], Nanyu Zhong[1,2,3,4], Wei You[5], Yanyan Zou[1,2,3,4*], Kunpeng Jian[1,2,3,4], Jiahuan Xu[1,2,3,4], Jian Sun[1,2,3,4], Baoxu Liu[1,2,3,4] and Wei Huo[1,2,3,4]

Abstract

Network function virtualization provides programmable in-network middlewares by leveraging virtualization technologies and commodity hardware and has gained popularity among all mainstream network device manufacturers. Yet it is challenging to apply coverage-guided fuzzing, one of the state-of-the-art vulnerability discovery approaches, to those virtualized network devices, due to inevitable integrity protection adopted by those devices. In this paper, we propose a coverage-guided fuzzing framework NDFuzz for virtualized network devices with a novel integrity protection bypassing method, which is able to distinguish processes of virtualized network devices from hypervisors with a carefully designed non-intrusive page global directory inference technique. We implement NDFuzz atop of two black-box fuzzers and evaluate NDFuzz with three representative network protocols, SNMP , DHCP and NTP , on nine popular virtualized network devices. NDFuzz obtains an average 36% coverage improvement in comparison with its black-box counterparts. NDFuzz discovers 2 0-Day vulnerabilities and 1 1-Day vulnerability with coverage guidance while the black-box fuzzer can find only one of them. All discovered vulnerabilities are confirmed by corresponding vendors.

Keywords: Coverage-guided fuzzing, Network devices, Network function virtualization

Introduction

Network function virtualization (Paper 2012) (NFV) provides programmable in-network middlewares to construct network services which are more robust and scalable. The virtualized network functions (VNFs) are built upon commodity hardware, e.g. x86 machine, and naturally reuse the fruitful innovations in their software stacks, e.g. operating systems. Thus, VNFs are essentially software implementations of network functions and can be deployed as an isolated virtual machine with the same functionality of their physical proprietary counterparts (Mijumbi et al. 2016; Han et al. 2015). Table 1 summaries 12 popular virtualized network devices across 9 mainstream vendors with 5 distinct functionalities (Column 3), showing a widespread application of NFV. As Table 1 shows, leading network equipment vendors, including Cisco (Line 2, 3), Juniper (Line 11, 12), Fortinet (Line 8) and more, are beginning to develop virtual network appliances based on mainstream open-source operating systems, e.g. Linux or Free BSD. shown in 'OS' Column. Therefore, a running VNF consists of a customized kernel and several userland processes which belong to different network functionalities.

*Correspondence: zouyanyan@iie.ac.cn
[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
Full list of author information is available at the end of the article

Table 1 Access control of popular virtualized network devices

Model	Vendor	Type	OS	N.Shell[a]	X.P.[b]
vEOS	Arista	Switch	Linux	✓	✓
CSR1000v	Cisco	Router	Linux	✗	✗
ASAv	Cisco	Firewall	Linux	✗	✗
SRA	Sonicwall	SSLVPN	Linux	✗	✗
SMA	Sonicwall	Gateway	Linux	✓	✓
PSA	Pulse Secure	SSLVPN	Linux	✗	✗
CHR	Mikrotik	Router	Linux	✗	✗
FortiGate	Fortinet	Firewall	Linux	✗	✗
SG6000	Hillstone	Firewall	Linux	✗	✗
VyOS	VyOS	Router	Linux	✓	✓
vSRX	Juniper	Firewall	FreeBSD	✓	✗
vMX	Juniper	Router	FreeBSD	✓	✗

[a]Normal Shell: whether the normal shell is supported

[b]Execution Permission: whether a third-party program can be executed

Unfortunately, the VNFs, together with their physical proprietary counterparts, are facing a high-security threat from vulnerabilities, such as CVE-2016-6366[1]of simple network management protocol (SNMP) , CVE-2017-3881[2] of Cluster Management Protocol (CMP) and CDPWN[3] of Cisco Discovery Protocol (CDP). They are harmful with all critical or high level according to CVSS 3.0 . Thus, researchers seek to introduce automated vulnerability discovery for VNFs. Coverage-guided fuzzing is one of the most efficient methods to discover vulnerabilities (Manèset al. 2019). Although numerous research work have been proposed to improve the effectiveness of coverage-guided fuzzing (Lyu et al. 2019; Yue et al. 2020; Böhme et al. 2017; Rawat et al. 2017; Gan et al. 2018), the main burden on applying such a fuzzing technique to the VNFs is to obtain the fine-grained runtime information of userland processes, e.g., their execution paths and executing basic blocks (trace), for the reason that a VNF is indeed a virtual machine with built-in integrity protection. Table 1 highlights the various integrity protection techniques ("N.Shell" and "X.P." columns) adopted by different vendors, 9 out of 12 devices are armed with at least one type of integrity protection technique. For example, CLI (Command-Line Interface) is the common interaction interface to VNFs. But, it is often a trimmed shell with customized commands provided by vendors for VNFs, which leads to inability to invoke a normal shell, e.g. bash and any third-parity software.

From the perspective of hypervisor (also known as virtual machine monitor, VMM), tracing userland processes is straightforward and has been well-studied as a part of virtual machine introspection (VMI) (Jain et al. 2014). Unfortunately, current VMI techniques can not deal with VNFs due to their *intrusive* manner: most of approaches require injecting code into guest virtual machines while integrity protection of VNFs disallows such injection. First, restriction of launching any third-party software makes approaches like kernel data structures reconstructing by drivers (Henderson et al. 2014; Yan and Yin 2012, https://libvmi.com/, https://github.com/Cisco-Talos/pyrebox) and code implanting (Carbone et al. 2012; Sharif et al. 2009) infeasible, as all of those methods require compiling and launching drivers inside guest virtual machines. Second, the diversity of operating systems for different VNFs and a lack of specific kernel configurations also make process outgrafting (Dolan-Gavitt et al. 2011; Fu and Lin 2012) and kernel data structures reconstructing by memory analysis (Socała and Cohen 2016), both of which rely on detailed kernel configuration to generate tools for snapshotting userland programs, not suitable for VNFs.

Challenge Tracing userland processes is a vital step in applying coverage-guided fuzzing to VNFs. The integrity protection commonly adopted by existing VNFs poses a critical challenge: how to trace a specific userland process of VNFs through the hypervisor in a *non-intrusive* way?

Our Approach We propose a coverage-guided fuzzing framework for VNFs with a non-intrusive process locating technique. This technique is implemented by inferring the *the correct page global directory (PGD) value of the target process* (target PGD for short) , which is widely treated as the process identification (Schumilo et al. 2017; Aschermann et al. 2019; Henderson et al. 2014, https://libvmi.com/, https://github.com/Cisco-Talos/pyrebox), from hypervisor view. Unlike traditional approaches limited by the integrity protection, we can infer target PGD with a lightweight differential analysis, which leverages two types of PGD-related information controlled by three controllable operations. The two types of side-channel information are enough to distinguish target PGD from others. Then we can extract the target process's running trace from the system's hybrid running traces with target PGD.

In this paper, a prototype of the coverage-guided fuzzing framework NDFuzz has been implemented. For the target process to be fuzzed, its PGD value can be inferred in a non-intrusive way. With target PGD, the target process can be located by the hypervisor. During fuzzing, NDFuzz collects the AFL-style bitmap of the target

[1] A critical vulnerability of Cisco ASA written by the Equation Group and leaked by the Shadow Brokers.

[2] A vulnerability for Cisco switches in Vault7, a large collection of documents of CIA released by WikiLeaks.

[3] A set of five vulnerabilities affecting Cisco equipment (https://www.armis.com/research/cdpwn/).

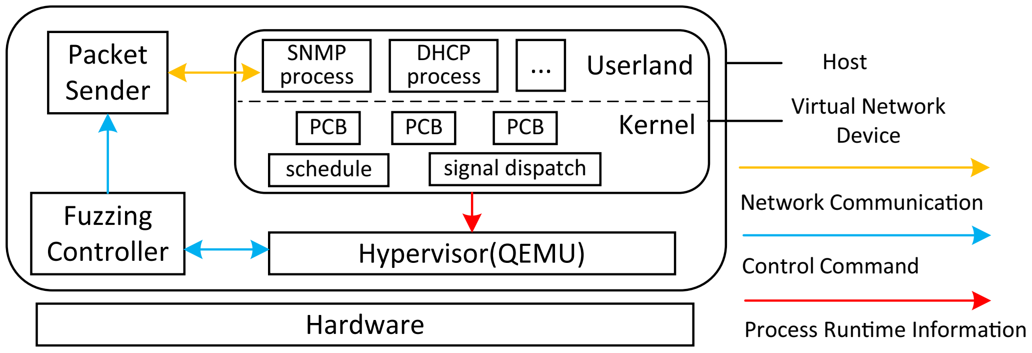


Fig. 1 Typical procedure to fuzz a network device

process and feeds the information back to guide the fuzzing. In order to monitor exceptional behaviors of target process, we provide a tracing-based in-VMM mechanism to catch the SIGSEGV signal and a coverage-jumping based out-of-VMM mechanism to deal with other exceptions.

We implement NDFuzz atop of two black-box fuzzers: a mature open-sourced fuzzer Mutiny (https://github.com/Cisco-Talos/mutiny-fuzzer) and a in-house fuzzer tailored for DHCP protocol, supporting greybox coverage-guided fuzzing in both variants. We leverage NDFuzz to fuzz three widely used protocols, SNMP , DHCP and NTP , on nine popular VNFs from seven vendors. Compared to its black-box counterparts, NDFuzz can notably improve the fuzzing performance of VNFs with at least an average 27 % coverage improvement. Furthermore, NDFuzz discovers 2 0-day vulnerabilities and 1 1-day vulnerability with coverage guidance while the black-box fuzzers can find only one of them. All discovered 3 vulnerabilities are confirmed by corresponding vendors.

This paper makes the following contributions:

- *Differential analysis based PGD inference technique for VNFs* The technique proposed can infer the PGD value of a given networking process of a VNF in a non-intrusive way. The key is to make the switching of target PGD value significantly different from other PGDs.
- *A coverage-guided fuzzing framework for VNFs* Leveraging target PGD, NDFuzz can obtain the runtime information of a userland process, and feed the information back to the fuzzer. Two monitoring mechanisms, i.e., a tracing-based in-VMM mechanism and a coverage-jumping based out-of-VMM mechanism, are provided to catch as many exceptions as possible. As a framework, NDFuzz can be used to augment black-box fuzzers. NDFuzz integrates a fuzzer which is developed by ourselves for DHCP. We also improve an existing general protocol fuzzer Mutiny (https://github.com/Cisco-Talos/mutiny-fuzzer) to support the coverage recording and guidance.
- *Vulnerabilities discovered in various VNFs* NDFuzz is leveraged to fuzz three protocols, SNMP, DHCP and NTP , in nine popular VNFs produced by seven vendors. In total, three vulnerabilities are confirmed by vendors, including two newly discovered vulnerabilities and one fixed vulnerability.

This paper is organized as follows: "Observation and motivation" section presents our observations and motivation to overcome the challenges. "Technical backgroud" section briefly introduces some crucial technical terms. "Overview" section shows the outline of NDFuzz containing two major phases which are introduced in the next two sections. "Phase I: target PGD inference" section explains the detail of PGD inference based on differential analysis. "Phase II: coverage-guided fuzzing" section describe the design and implementation of the fuzzing process. "Experiment and evaluation" section shows the result of NDFuzz. The shortcoming and future work are discussed in "Discussion" section. "Related Work" section lists the related work. Finally, "Conclusion" section summarizes this paper.

Observation and motivation

Figure 1 illustrates a typical fuzzing procedure for a virtualized network device running as a virtual machine containing several processes inside. Different processes handle different protocols, each with a corresponding process control block (PCB) in kernel space. The process is scheduled as a normal Linux kernel does. The kernel also dispatches exception signals such as SIGSEGV to the corresponding process when the process crashes.

Because of the integrity protections of VNFs, black-box fuzzing is one of the most straightforward ways to discover vulnerabilities, which is widely used in IoT devices fuzzing (Feng et al. 2021; Chen et al. 2018; Zhang et al. 2019). The main fuzzing function is implemented in the packet sender, which can send a mutating request to the target process (yellow arrow in Fig. 1). Once the communication is interrupted or responded with specific contents, a crash may occur, and the request is recorded. However, there are three limitations about the blackbox fuzzing: (1) The performance, such as code coverage, is hard to be evaluated. (2) The mutation is inefficient without runtime information guidance such as coverage. (3) The response-based monitoring (also known as liveness-check Muench et al. 2018) mechanism is quite unreliable. Although the coverage-guided fuzzing is widely used in the software to overcome those shortcomings, it can not be employed for VNFs directly.

For coverage-guided fuzzing (greybox fuzzing), a fuzzing controller is introduced to guide the mutation according to the request-related coverage. However, the coverage of the VNFs is not easy to obtain because of the integrity checks. The potential way is to implant a program into the virtualized network devices to monitor the runtime information (Gao et al. 2020). The method requires two preconditions unsupported by the VNFs, i.e., the ability to implant a program and the permission to execute the program.

The only general way is to obtain code coverage in a non-intrusive way by leveraging the hypervisor. The non-intrusive way means *no modification of the original VNF, no shell permission needed and no agent placed*. Under this constraints, some hypervisor-assist works such as TriforceAFL (https://github.com/nccgroup/TriforceAFL), kAFL (Schumilo et al. 2017), REDQUEEN (Aschermann et al. 2019) and FirmAFL (Zheng et al. 2019) all cannot work.

From the hypervisor view, all runtime information of the guest operating system, such as instructions, registers and memory can be obtained directly. However, the information is just raw data without semantic information, which is the well-known semantic gap problem (Dolan-Gavitt et al. 2011). As described in Introduction, numerous works to solve the semantic gap with VMI technique are limited in our fuzzing scenario. As Table 1 depicts, numerous VNFs are developed atop of the modern multi-tasking operating system (Linux, FreeBSD, etc.). Therefore different network services always run as different daemon processes and our fuzzing target is indeed the userland process of a VNF. The only thing we need for coverage guidance is the trace. Thus the major problem is solved once we can filter the correct trace of the target process from hybrid raw data from hypervisor view. Page global directory (PGD) of a process is suitable for distinguishing different processes in our scenario with three features: (1) For the paging mechanism for a process, both Linux and FreeBSD adopt a multilevel paging model. The top-level is PGD, a physical page frame that is unique for different processes and can be treated as a process identification. (2) Unlike the target (i.e. image parser) whose lifecycle is just a round of parsing in mainstream fuzzing (Gan et al. 2018, http://lcamtuf.coredump.cx/afl/, Lyu et al. 2019; Aschermann et al. 2019), the network process always acts as a daemon process and handles network requests persistently in a single process. Obviously, PGD value is constant unless the process crashes or restarts. (3) When the kernel schedules a process, PGD is loaded into *a special register* (CR3 of x86 and x86_64). The change of this register can be treated as a specific event of the hypervisor. In other words, once the PGD value of the target process is known, the hypervisor can know whether the target process is scheduled and running by monitoring the change of the CR3 register.

These observations reveals that we can trace the target network service, a daemon process, in a non-intrusive way through obtaining the correct PGD values of the target process. NDFuzz focuses on the network service satisfying three characters: (1) Its function is independent and works as a single daemon process. (2) The protocol is whole or partial stateless so that the requests are unrelated to each other. (3) The service is deployed in the C/S architecture, and the process works as a server while the fuzzer is the client.

Technical backgroud

In this section, we briefly introduce some crucial technical terms of this paper.

QEMU

QEMU is a hosted virtual machine monitor with JIT compilation that translates the target's code to native instructions and executes at native speed. Tiny code generator (TCG) works by translating the guest basic block into an architecture-independent intermediate representation (IR), then the backend lowers the IR into native host instructions. There are two general modes as user mode and system mode. *The user mode* can execute a single process while *the system mode* emulates a full operating system (Virtual Machine, VM), including a processor and various peripherals. QEMU machine protocol[4] (QMP) is used for system mode to control the virtual machine such as save/load a snapshot, dump memory, etc. In fuzzing, QEMU user mode is always used for single binary program fuzzing scenario such as

[4] https://wiki.qemu.org/Documentation/QMP.

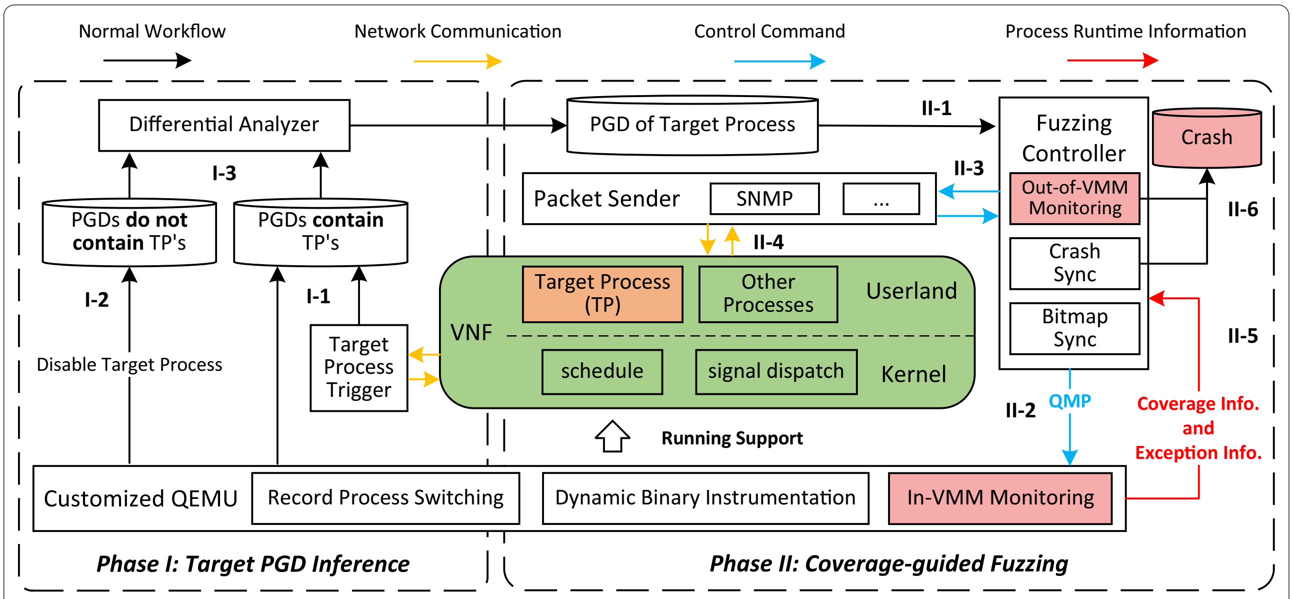


Fig. 2 Overview of NDFuzz

QEMU mode of AFL, system mode is always used for the target such as kernel which needs a whole system.

Network services

Simple network management protocol (SNMP) is an Internet Standard protocol used for the management and monitoring of network-connected devices. It is widely used in network management for network monitoring by different object identifiers (OID), the representation of device function.

Dynamic host configuration protocol (DHCP) is a client/server protocol for automatically assigning IP addresses and other communication parameters to devices connected to the network. DHCP is widely used in devices that have a LAN for connecting such as a router, gateway, firewall and more.

Network time protocol (NTP) is designed for Time synchronization between different network nodes over packet-switched, variable-latency data networks. It can provide high precision time correction for other clients in the network and keep the time consistency in the network.

Overview

Figure 2 describes the high-level overview of our fuzzing framework NDFuzz. It consists of two critical phases: target PGD inference phase (Phase I) depending on the differential analysis of process switching to obtain the identification of target process, actually the PGD value of a process. Then phase II uses this PGD value to drive the coverage-guided fuzzing. Since the PGD is always changed when the process or system restarts, we take a snapshot for each given VNF image to avoid redundant work. For a given VNF image, a snapshot with the PGD value of the target process can be treated as a *fuzzing instance*. The fuzzing instance can be copied and deployed independently for different aims.

Phase I treats a running VNF as the input supported by the customized QEMU, the virtual machine monitor (VMM or hypervisor). Although the customized QEMU can obtain all runtime information of a VNF, the semantic information is still unknown. Fortunately, we only need is the PGD value of the target process (TP) to drive the coverage-guided fuzzing. We record a set of PGD values of alive processes and their process switching times. With the two attributes, we can leverage a lightweight differential analysis to infer the correct PGD value of TP. In this side-channel way, we can automatically locate TP accurately and efficiently. The main steps are shown in the left half of Fig. 2. *I-1:* The customized hypervisor record a list of PGD values and the scheduled times. In order to ensure that TP will be scheduled, we continuously make requests to TP during PGD recording. *I-2:* We disable or restart TP and another list of PGD values and schedule times are recorded. *I-3:* A lightweight differential analysis is applied to the two lists of PGD values to obtain the target PGD.

Phase II is depicted in the right half of Fig. 2. During the fuzzing procedure, the fuzzing controller coordinates the customized QEMU and the packet sender with control command. Fuzzing controller loads the configuration of target process (*II-1*), initializes customized QEMU (*II-2*) and the packet sender (*II-3*). With the PGD value of the target process, customized QEMU can know if TP is scheduled by monitoring the change of CR3 register in x86 or x86_64 architecture. Therefore, the trace of the target process can be recorded when TP is running. The recording is related to the request handling. Before the packet sender sends a mutated request (*II-4*) , the fuzzing controller notifies QEMU to enable the trace recording. Once the request is handled completely, the fuzzing controller disables recording and obtains the traces for coverage guidance from customized QEMU through QMP protocol and shared memory (*II-5*) . The customized QEMU can record AFL-style bitmap in a normal running loop. Once a crash occurs, we can monitor it with two mechanisms (*II-6*) . An in-VMM monitoring mechanism can catch the exception signal, such as SIGSEGV, accuracy through partial kernel tracing. Moreover, we also developed an out-of-VMM monitoring mechanism based on coverage-jumping to catch the exceptions missed by the in-VMM method. After recording the exceptional request, fuzzing controller loads the initial snapshot of VNF to continue the fuzzing.

The whole procedure is entirely transparent for VNF because there is no modification of VNF, no shell permission needed and no agent program placed. It is a non-intrusive way to turn a black-box fuzzing into a gray-box fuzzing for VNF.

Phase I: target PGD inference

In this section, we illustrate the detailed implementation of process locating with the help of differential analysis. Due to the environment limitation of VNFs, it is difficult to obtain the specific kernel offset needed by mainstream VMI and memory analysis techniques. Thus, we cannot traditionally obtain the mapping of processes and their PGD values. However, our aim is different from the VMI. Instead of obtaining numerous information belonging to the process control block (PCB), we only need the PGD value of the target process to identify the target process. Although the VNF has several inherent limitations, we can still control part of the internal VNF such as the status of a network process.

Firstly, through CLI or other management methods (e.g. web interface), we can enable, disable or restart a network service[5], and the network service is related to a userland process which is corresponding to a specific PGD value. Specifically, enabling a network service binds a PGD value and the process. Disabling or restarting[6] the service will eliminate this relation. *Secondly*, from the view of hypervisor, all PGD values of the userland processes can be obtained by monitoring the switching of CR3 register. In a given period, we can not only record what PGD values are loaded into CR3 that reveals how many processes are switched, but the time of each PGD value that reveals the execution frequency of a PGD of process. *Thirdly*, the network service always acts as the daemon process that stays in a loop like "waiting for a request—handling the request—waiting for the next request". When a request reaches the VNF from outside, the kernel will wake up the corresponding process according to the port, and the process switching occurs. *Finally*, we can configure the VNF through CLI to create a "stable" environment. This environment consists of the target process, as few as unrelated processes by disabling them, and the system services that cannot be configured.

These four points are corresponding to two types of side-channel information and three operations to change them. By controlling the binding (operation 1) and eliminating (operation 2) of the relation of PGD value of target process, we can obtain a set of PGD values containing the PGD value of target process and another set do not contain (information 1). By creating network requests to the target process (operation 3), we can increase its schedule times, which equals the process switching time (information 2). Thus operation 3 makes the switching time of target PGD be quite more than the default. With the well-configured environment, this method can significantly differ from other processes on process switching with less noise. Therefore, these operations and information are enough to ensure the correct PGD value of the target process through differential analysis.

[5] Note that these operations might be executed more directly with the normal shell.

[6] The PGD value is bound to a process, once a network service is restarted, the process is actually changed, so the PGD is always changed too.

Algorithm 1: Process Locating by Differential Analysis.

```
Input: A running VNF $V$, a method $trigger(T_V)$to request the target process $T_V$.
Output: A fuzzing instance containing $V_{snapshot}$ and PGD value $P$ of target process $T_V$ .
1  $binding(T_V)$;
2  $V_{snapshot} \leftarrow take\_snapshot(V)$ ;
3  $PGDs \leftarrow \{\}$ ;
4  while $process\_switch\_time < N$ do
5            ▷ Note this trigger running is concurrent with process switching recording ;
6      $trigger(T_V)$ ;
7      $PGD\_value \leftarrow monitor\_CR3\_switching(V)$ ;
8      if $PGDs.has\_key(PGD\_value)$ then
9          $PGDs[PGD\_value] \leftarrow PGDs[PGD\_value] + 1$ ;
10     else
11         $PGDs[PGD\_value] \leftarrow 1$ ;
12 $eliminating(T_V)$;
13 $PGDs\_without\_target \leftarrow \{\}$ ;
14 while $process\_switch\_time < N$ do
15     $PGD\_value \leftarrow monitor\_CR3\_switching(V)$ ;
16     if $PGDs\_without\_target.has\_key(PGD\_value)$ then
17         $PGDs\_without\_target[PGD\_value] \leftarrow PGDs\_without\_target[PGD\_value] + 1$ ;
18     else
19         $PGDs\_without\_target[PGD\_value] \leftarrow 1$ ;
20 $PGD\_set \leftarrow set(PGDs) \setminus set(PGDs\_without\_target)$;
21 $P \leftarrow select\_PGD\_value\_with\_most\_switching(PGD\_set, PGDs)$;
22 return $P$, $V_{snapshot}$ ;
```

Algorithm 1 reveals how to infer PGD value of the target process T_V of a VNF V . We define operations *binding*(T_V) and *eliminating*(T_V) to represent operation 1 and operation 2 with correct configuration, and *trigger*(T_V) means a network request to target process T_V from outside of VNF (operation 3). We first bind T_V with a PGD value by enabling the service and taking a snapshot for later fuzzing. As described in "Phase I: target PGD inference", once T_V is running and triggered by a network request, it will be scheduled. Therefore we use the hypervisor to record the process switching N times (we always use 1000) and trigger the process by network simultaneously. Then we obtain a list *PGDs* containing all PGD values with each process switching time. Because of our strategy, the target PGD value must be in *PGDs* with a top process switching times. Then we eliminate the relation of T_V and target PGD by disabling or restarting the service. We can similarly obtain the list *PGDs_without_target*. Next, we transform the *PGDs* and *PGDs_without_target* into sets of PGD values and obtain the difference set PGD_{set}. Finally, we compare the elements which have the same PGD values of PGD_{set} and *PGDs* list, and the PGD value with the most scheduled times is the PGD values of the target process.

This method to locate the process is general for various multi-tasking operating systems such as Linux and FreeBSD. Obviously, the versions are hardly affected because it is a general design of the operating system. In x86 and x86_64 architecture, the CR3 register saves the PGD values and the instructions to read or write this register are specific. Although we cannot map all processes and PGD values through memory analysis which depends on offsets or debug information, this method is enough for our scenario.

We customize QEMU as our hypervisor to record the PGD values by hooking the CR3 related instructions. The implement of *trigger*(T_V) is related to the protocol of T_V. The trigger is general for the target process of different VNF with the same protocol. The customization is needed for a target process with a new protocol. Implementing a trigger is simple because it can just send a packet to the correct port of VNF. The packet can be captured from normal communication or crafted according to the RFCs.

Phase II: coverage-guided Fuzzing

With the process control information, we can trace and monitor a userland process via hypervisor and collect the runtime information to guide the fuzzing. In this section, we depicts the implementation of the fuzzing phase. "Driving the whole fuzzing" section illustrates the whole procedure of VNF fuzzing. "Monitoring a userland

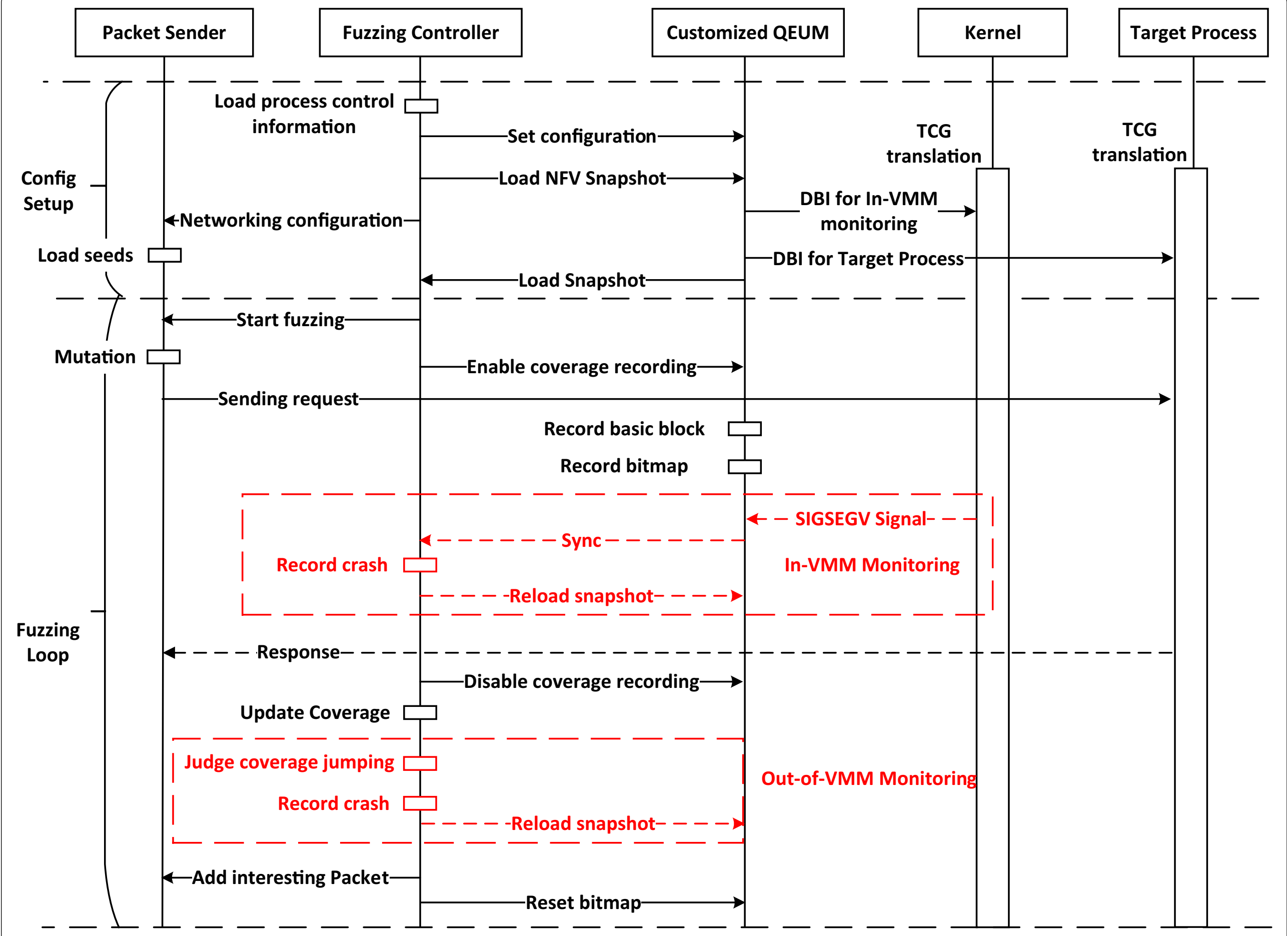


Fig. 3 Overview of the fuzzing interaction

program" section reveals how to monitor a crash from hypervisor view. "Customized hypervisor" section shows the implementation of the customized QEMU.

Driving the whole fuzzing

Figure 3 shows the whole fuzzing interaction of NDFuzz. Fuzzing controller drives the whole procedure by controlling customized QEMU and packet sender. Customized QEMU executes the dynamic binary instrumentation for monitoring exceptions and tracing the target process. Packet sender crafts the packet and sends them by notification of fuzz controller. During the handling of a packet, QEMU record the real-time information. Fuzzing controller determines the start and end of bitmap and tracing recording by the response from the userland program with an acceptable timeout limitation. Then bitmap information of each request is synchronized by shared memory through customized QMP command. According to bitmap, the interesting case is put into the priority queue as guidance. At the same time, it synchronizes with QEMU about signal monitor and analysis the coverage for coverage monitor. Once an exception occurs, fuzzing controller saves the corresponding information, then reloads the snapshot for later fuzzing. All command and control operations are transparent to the device atop the hypervisor, so the device treats all requests as usual.

We design an intermediate layer packet sender to be compatible with different customized protocols fuzzing, including the mutation interface used in the mutation thread and the sending interface used in the communication thread. The mutation thread is used to mutate the requests to satisfy the protocol specification and maintain the testing queue, and the communication thread communicates to the target process according to the rules and maintains the interesting queue. The mutation thread takes requests from the interesting queue and puts them into testing queue after mutation. The communication thread stores testcases by coverage growth into the interesting queue to support coverage guidance.

Besides customized protocols integrating, we also implemented the core functionality of fuzzing controller as APIs to support the adaptation of the existing black-box network fuzzer. A black-box fuzzer can easily obtain coverage to evaluate performance even support the coverage-guided fuzzing.

Monitoring a userland program

Tracing a userland program can provide the coverage information that drives the graybox fuzzing. Once an exception occurs, NDFuzz needs to know the termination and record the related testcase. Note that monitoring should also be non-intrusive. Liveness-check (Muench et al. 2018) should work for part of our scenario but with apparent shortcomings. To monitor the exceptions through the hypervisor view, we develop two monitoring mechanisms by tracing the exceptional handler and observing the coverage jumping of the userland process.

In-VMM Monitoring According to Muench's research (Muench et al. 2018), segment tracking can archive 80% types of artificial vulnerabilities discovered in their experiment. For Linux and FreeBSD, detecting Segmentation Fault (SIGSEGV) equals this technique. When a userland program triggers an access violation, the kernel will take over and notify the process with a SIGSEGV signal. For hypervisor, tracing the handler execution during the target process execution is a general in-VMM monitoring mechanism. Rule 1 indicates the in-VMM monitoring. The target PGD is known and the process handler is specific for different kernels. We locate the SIGSEGV handler by finding the kernel basic block address which contains some special strings access. For Linux, when a userspace program triggers SIGSEGV, kernel logs the exception information by calling printk() function with the format string "%s%s[%d]: segfault at ". For FreeBSD, the feature string is "(core dumped)". The kernel binary is not difficult to get by mounting the VNF images.

$$\begin{cases} PC_{VMM} = SIGSEGV_{kernel} \\ CR3_{VMM} = PGD_{target\_process} \end{cases} \tag{1}$$

Out-of-VMM Monitoring Although monitoring SIGSEGV signal can cover most of the process crashes, some other exceptional situations can also cause process crash or termination, such as the crash by SIGABRT signal triggered by heap errors or restarting by a watchdog. Because the handle of the SIGABRT signal is quite more complex than SIGSEGV, it is not suitable to be caught in the In-VMM monitoring. In this situation, monitoring only by SIGSEGV signal might miss some exceptional testcases. However, if we ignore these false negatives, the whole graybox fuzzing which is related to the target PGD value will be broken. For example, once the target process is crashed by SIGABRT signal, the In-VMM mechanism will miss it but the watchdog might restart this process to recover its running. After the process restarting, it can handle the network request as before but its PGD value has changed because of the restarting. Because NDFuzz uses target PGD value to obtain the runtime information of target process, the PGD change caused by false negatives will make the whole fuzzing abnormal and is quite unacceptable.

Therefore, we also provide an additional monitoring mechanism with code coverage based on an ingenious observation of our userland program tracing. As described in "Phase I: target PGD inference" section, the PGD value never changed during the normal running of the target process and QEMU records the coverage information with the filter of this PGD value. So once the process is terminated, modified QEMU cannot trace anything with the previous CR3 value. This causes a "coverage jumping" from non-zero to zero and reveals the exception that was missed by in-VMM monitoring. Out-of-VMM monitoring might catch some terminations which are not caused by a vulnerability in a few extreme situations, but we think these unexpected exits are also should be caught to ensure that the whole fuzzing works well.

Customized hypervisor

To support the graybox fuzzing, we developed the customized QEMU based on QEMU-TCG. A fuzzing job can be deployed on just a PC or cloud with this low-level implementation. We hook the CR3 read and write operation of QEMU to know whether the target process is scheduled. We also modify the TCG engine for dynamic binary instrumentation and provide AFL-style bitmap for guidance (edge coverage). There could be three types of instruction during the userland process tracing: instruction of correct userland process, instruction of others userland processes, and kernel instructions. We use the PGD value of the target process to filter the correct userland process. Then we use the RIP or EIP value to filter the instructions of kernel space and userspace.

A straightforward way to dynamic instrument is to hook the TCG run-loop of cpu_exec() function of qemu/accel/tcg/cpu-exec.c file. However, it will bring a huge performance overhead by disabling the block chaining feature. Compared to the native code execution, the translation is expensive and translation blocks (TBs) are saved in the TCG cache in normal QEMU execution. When the TB is executed, QEMU will find the next basic block, and it also brings the extra performance overhead. Therefore QEMU provides the block chaining to link the adjacent TBs, such as a direct jump with the known destination address. Without block chaining optimization, each basic block will be translated expensively repetitively without cache. For our system-mode QEMU, the cost is quite unacceptable. AFL++ (Fioraldi et al. 2020, https://andreafioraldi.github.io/articles/2019/07/20/

Table 2 Code size of fuzzing components

Fuzzer	Mutiny			ZDHCP	NDFuzz	NDFuzz [a]	QEMU
Type	ORI	NFB	F	NFB	–	–	–
LoC	2.0k	2.7k	2.8k	4.6k	3.6k	6.2k	+1.8k (C)

[a]Contains well-adapted ZDHCP-FB

aflpp-qemu-compcov.html) optimized this by inserting the bitmap calculation into the translation block and got 3x–4x speedup.

Inspired by this optimization, we directly insert our helper function into the translation block for tracing and monitoring. The bitmap is saved in shared memory which is reset before each fuzzing iteration. The corresponding byte of the bitmap is incremented when an edge transition occurs. The implementation of monitoring is a little different from tracing. Because it focuses on monitoring the execution of the exception handler which is related to the address of a specific basic block, we insert the helper function when this basic block is translated. Therefore when the basic block is executed, the helper function will mark the signal to achieve the monitoring.

Experiment and evaluation

In this section, we evaluate the prototype implementation of NDFuzz. In short, we would like to answer the following research questions:

1 *Generality* Does our target PGD inference technique work well on different VNFs?
2 *Coverage improvement* How much the coverage has been improved with coverage-guidance?
3 *Vulnerability discovery* How effective is NDFuzz in finding the real vulnerabilities in VNFs?
4 *Overhead* What is the overhead of NDFuzz about instrumentation?
5 *Case study* How does coverage guide the vulnerability discovery?

Experiment Setup We deployed our fuzzing on a Ubuntu 20.04 LTS with Intel(R) Xeon(R) Gold 6242R CPU @ 3.10GHz and 120GB memory. To avoid the potential noise among different devices (traffic of ARP, NDP, etc.) that might affect the fuzzing, we performed network isolation for each fuzzing instance. More specifically, we prepare a whole virtual machine for one fuzzing instance and NDFuzz deployment. Each virtual machine is given 8 GB memory and 50 GB virtual disk. The whole process from phase I to phase II is automatic with the only manual effort of device configuration. Due to the significant difference among devices, it took us about 3 h per device to complete the configuration of the SNMP (version 2c) , DHCP and NTP services to prepare for a fuzzing instance.

In order to verify the performance of NDFuzz in improving the coverage and finding crashes, we adopt four fuzzing engines which contain 2 general fuzzing engines (Mutiny-No-Feedback and Mutiny-Feedback) and 2 protocol customized fuzzing engines (ZDHCP-No-Feedback and ZDHCP-Feedback). We totally tested 18 fuzzing instances and each of them has been tested by several fuzzing engines by copying them. The total CPU hours of fuzzing is 3168 (9*2*72+4*4*72+5*2*72) h.

Code Size of NDFuzz Table 2 shows the statistic of NDFuzz. We added about 1.8k lines of C code to customize QEMU for our requirement. Original Mutiny is about 2.0k lines of python code and we add 0.7k which contains the API of NDFuzz to enable the runtime information obtaining. For Mutiny-FB mode, we added about 0.8 k lines of Python on the original Mutiny. The ZDHCP-NFB is about 4.6 k lines of Python code with the NDFuzz API to record the coverage in black box fuzzing. The total NDFuzz is 6.2 k line of Python with ZDHCP-FB integrated.

Generality for different devices

In order to demonstrate the generality of NDFuzz, we selected nine real-world VNFs as fuzzing targets among seven famous vendors (Arista, Juniper, SonicWALL, Fortinet, Hillstone, Pulsesecure and Vyos). These devices belong to Switch, Router, Firewall, Gateway, and SSLVPN 5 types. Table 3 illustrates the detailed information of our nine target VNFs. We can see 77.8% (7 of 9) devices are based on Linux of various versions except for Juniper which is based on FreeBSD. The kernel versions of Linux are all different with a 11-year gap from 2.6.32 to 4.19.142. One of the seven Linux-based VNFs is x86 and others are x86_64. Although the two architectures are similar, they are still different such as calling convention. However, our lightweight target PGD inference technique can obtain the correct target PGD among different OS and versions. This is because the process scheduling mechanisms are general for various multi-tasking operating systems.

We selected SNMP, DHCP and NTP protocol as target service. SNMP is a typical stateless protocol that is used to manage a device according to different OID nodes.

Table 3 Virtualized network devices and services for fuzzing

Model	Type	OS	Arch	Kernel version	Fuzzing protocol		
					SNMP	DHCP	NTP
vEOS	Switch	Linux	x64	4.19.142	Yes	No	Yes
PSA	SSLVPN	Linux	x64	2.6.32	Yes	No	No
SRA	SSLVPN	Linux	x86	3.1.0	Yes	No	No
FortiGate	Firewall	Linux	x64	3.2.16	Yes	Yes	Yes
SG6000	Firewall	Linux	x64	3.10.20	Yes	Yes	No
SMA	Gateway	Linux	x64	4.4.12	Yes	No	No
VyOS	Router	Linux	x64	3.13.11	Yes	Yes	Yes
vSRX	Firewall	FreeBSD	x64	11.0	Yes	Yes	Yes
vMX	Router	FreeBSD	x64	11.0	Yes	No[a]	Yes

[a]We failed to enable its DHCP Server of vMX although have tried our best to configure it

Table 4 Statistic of proportions and ranks of different processes

Model	SNMP-T[a]		SNMP-N[b]		DHCP-T		DHCP-N		NTP-T		NTP-N	
	Prop. (%)	Rank	Prop. (%)	Rank	Prop.	Rank	Prop.	Rank	Ratio	Rank	Ratio	Rank
vEOS	30.9	1	2.3	7	N/A	N/A	N/A	N/A	35.7%	1	0.7%	15
PSA	58.0	1	0.3	37	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SRA	27.3	2	7.0	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
FortiGate	52.6	1	3.1	9	35.5%	1	0.30%	50	31.3%	1	0.3%	23
SG6000	22.9	2	0.2	15	20.6%	2	0.20%	24	N/A	N/A	N/A	N/A
SMA	16.0	2	0.9	19	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
VyOS	54.0	1	16.0	7	27.5%	1	1.20%	15	31.9%	1	2%	7
vSRX	14.8	2	1.4	11	20.0%	2	1.5%	9	29.1%	2	0.2%	22
vMX	29.1	2	1.10	11	N/A	N/A	N/A	N/A	43.6%	2	0.6%	19

[a] *T* suffix means -Trigger, shows the ratio and rank of switching times of target PGD during the 1000 times process switching with trigger

[b] *N* suffix means -Normal, shows the ratio and rank of switching times of target PGD during the 1000 times process switching in normal running

DHCP is a little different because it contains the client and server side which is our fuzzing target. We also treat it stateless to focus on option parsing by fixing the IP address and marking each request with a unique session ID. NTP is simpler than SNMP and DHCP and is used for clock synchronization between different network nodes. The support of the three protocols in different devices is listed in the column 6–8 of Table 3. All of the devices can obtain target PGD in less than 20 s. This is efficient and negligible compared to the 72 h fuzzing.

We counted the proportion and rank for illustrating phase I and the detailed information are listed in Table 4 . We first got the target PGD value through differential analysis of phase I. Then, we loaded the snapshot and counted the default switching times according to the correct PGD value. The percentage reveals the proportion of switching times of target PGD in total one thousand times switching. SNMP-T, DHCP-T and NTP-T are the statistic of target PGD with trigger operation. SNMP-N, DHCP-N and NTP-N shows statistic during the normal VNF running. From SNMP-N, DHCP-N and NTP-N, we can see that SNMP, DHCP or NTP processes are usually scheduled with a small proportion. SNMP-T, DHCP-T and NTP-T shows that these processes are always scheduled in a significantly larger proportion and top rank with the trigger operation . Among the 9 VNFs, we totally obtained the information of 18 PGD values, including 9 for SNMP, 4 for DHCP and 5 for NTP . With trigger operation, 9 of 18 take the first place and others are second. This is because some system processes might take a higher priority than target processes.

The mainstream way to obtain the PGD value is through the memory analysis (https://libvmi.com/, https://github.com/Cisco-Talos/pyrebox, Henderson et al. 2014; Yan and Yin 2012, https://github.com/volatilityfoundation/volatility), but on the premise of

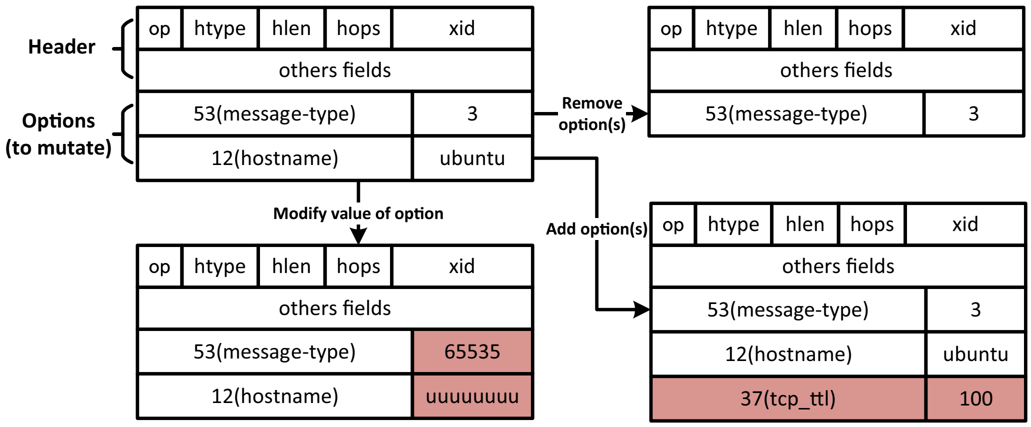


Fig. 4 The core design of ZDHCP

non-intrusive, we cannot obtain the proper input in their way such as kernel data structures reconstruction by drivers and more. Thus this potential way is another question to be solved.

Our differential analysis solution can only infer the PGD and cannot obtain detailed information such as the process name which is related to this PGD value, while the potential way can obtain the accuracy content. This is the major disadvantage. On the other hand, the potential way needs to parse each kernel file to extract the needed information and then parse the whole memory snapshot, which is complex with more steps than our solution . This is the advantage of our solution .

Improvement of code coverage

We fuzzed all VNFs with a significantly lower cost of both fuzzing target and time with the help of virtualization than physical devices. The fuzzing engines are illustrated as follows:

Mutiny-No-Feedback (Mutiny-NFB) Mutiny (https://github.com/Cisco-Talos/mutiny-fuzzer) is a network fuzzing framework that mutates the requests of raw PCAPs. It is developed by Cisco Talos security team and uses Radamsa (https://gitlab.com/akihe/radamsa) to perform mutations. As a mutation-based fuzzer, it treats the PCAP file as input and generates a fuzzing template according to the contents. Thus Mutiny is general for TCP or UDP protocols and is designed for black-box protocol fuzzing so that it can test VNFs. We select it as a benchmark to show the performance of block-box fuzzing. We adapt Mutiny with our APIs to record the coverage information without modifying the original fuzzing logic. It can test all target protocols (SNMP, DHCP and NTP).

Mutiny-Feedback (Mutiny-FB) Based on coverage information, we further modified Mutiny with the coverage guidance support. Actually, we added a priority queue to save the interesting requests to guide the mutation. If the queue is empty, it will execute the original fuzzing logic.

ZDHCP-No-Feedback (ZDHCP-NFB) DHCP is a popular protocol of devices that use different options to indicate the function of a request. The DHCP client requests different information via various options from DHCP server. Then DHCP server resolves the options and responses to the client. One DHCP request can contain several different options described in RFC 2132 and more. Therefore, we designed and implemented ZDHCP to fouces on mutation and combination of options instead of the completely random mutation as Mutiny. Figure 4 depicts the core idea of ZDHCP. In our implementation of ZDHCP, we use the 119 options implemented in scapy (https://scapy.net/) to help us to build a legal DHCP packet. ZDHCP works as a client to fuzz the DHCP server and as a black-box fuzzer which is also adapted with APIs of NDFuzz to record coverage of target processes.

ZDHCP-Feedback (ZDHCP-FB) As described in subsection Driving the Whole Fuzzing, we design an intermediate layer to be compatible with different protocol fuzzer implementations. The implementation of the mutation interface and sending interface is built in ZDHCP. Therefore, it is easy to integrate ZDHCP into NDFuzz. We use the coverage to guide the mutation of options.

For each VNF that supports SNMP, we deploy Mutiny-FB, Mutiny-NFB with the same seeds for at least 72 h. The seed of SNMP is related to the OIDs, which is obtained by a standard program *snmpwalk* (https://linux.die.net/man/1/snmpwalk). Snmpwalk uses SNMP_GET-NEXT requests to retrieve a subtree of management values. We use it to retrieve the root node and obtain all OIDs of a VNF. Then we convert the OIDs into the

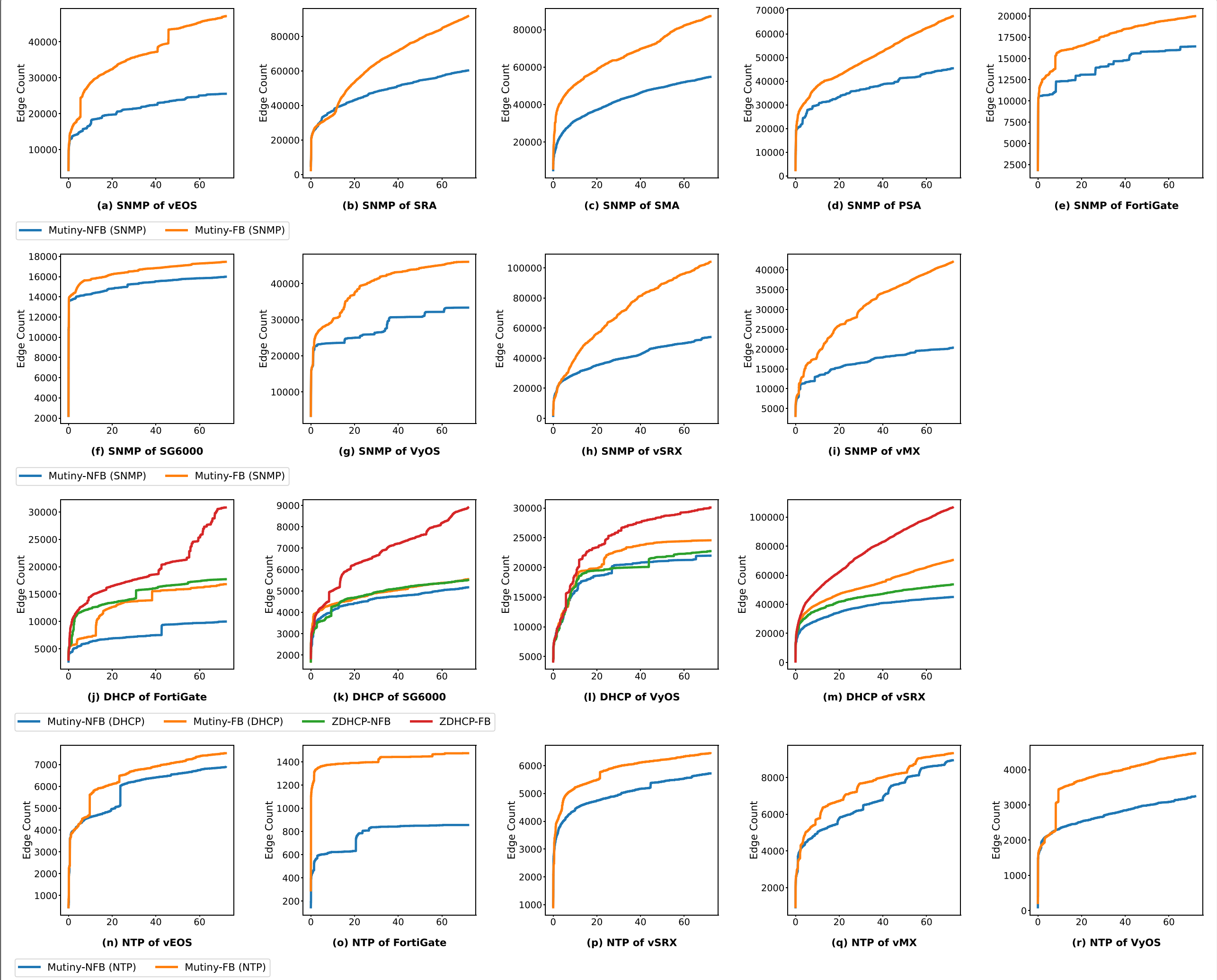


Fig. 5 Edge count trend of the 9 SNMP, 4 DHCP and 5 NTP fuzzing instances for 72 h

format of Mutiny, the PCAP files. For DHCP, we deploy Mutiny-FB , Mutiny-NFB, ZDHCP-NFB and ZDHCP-FB with the same seeds. Because we do not find a program similar to snmpwalk, we get the seed of DHCP in 2 ways. One is to collect requests during the normal communication, the other one is to generate requests according to the specification, then filter the request which the DHCP server can reply.

Figure 5 depicts the edge trend during the 72h fuzzing of each fuzzing instance. The subfigure (a) to (i) are of SNMP , the (j) to (m) are of DHCP and the (n) to (r) are of NTP. Figure 6 shows the edge improvement about the different fuzzing engines for the same fuzzing instance. For SNMP (Fig. 6a), the comparison of Mutiny is intuitive and straightforward, Mutiny-FB performs an average of 56.93% more edge coverage than Mutiny-NFB from 9.22% of SG6000 to 106.33% of vMX.

For DHCP (Fig. 6b), the comparison is more complex because there are four fuzzing engines. To compare the result in different aspects, we calculate four types of comparison as follows:

1 *Mutiny-FB vs. Mutiny-NFB and ZDHCP-FB vs. ZDHCP-NFB* Mutiny-FB has an average of 36.03% improvement than Mutiny-NFB from 7.33% of SG6000 to 68.32% of FortiGate. ZDHCP-FB has an average of 66.58% improvement than ZDHCP-NFB from 32.28% of VyOS to 98.56% of vMX of ZDHCP. Together with the result of SNMP, these comparisons prove the coverage guidance can indeed improve the edge for different protocols and fuzzing engines.

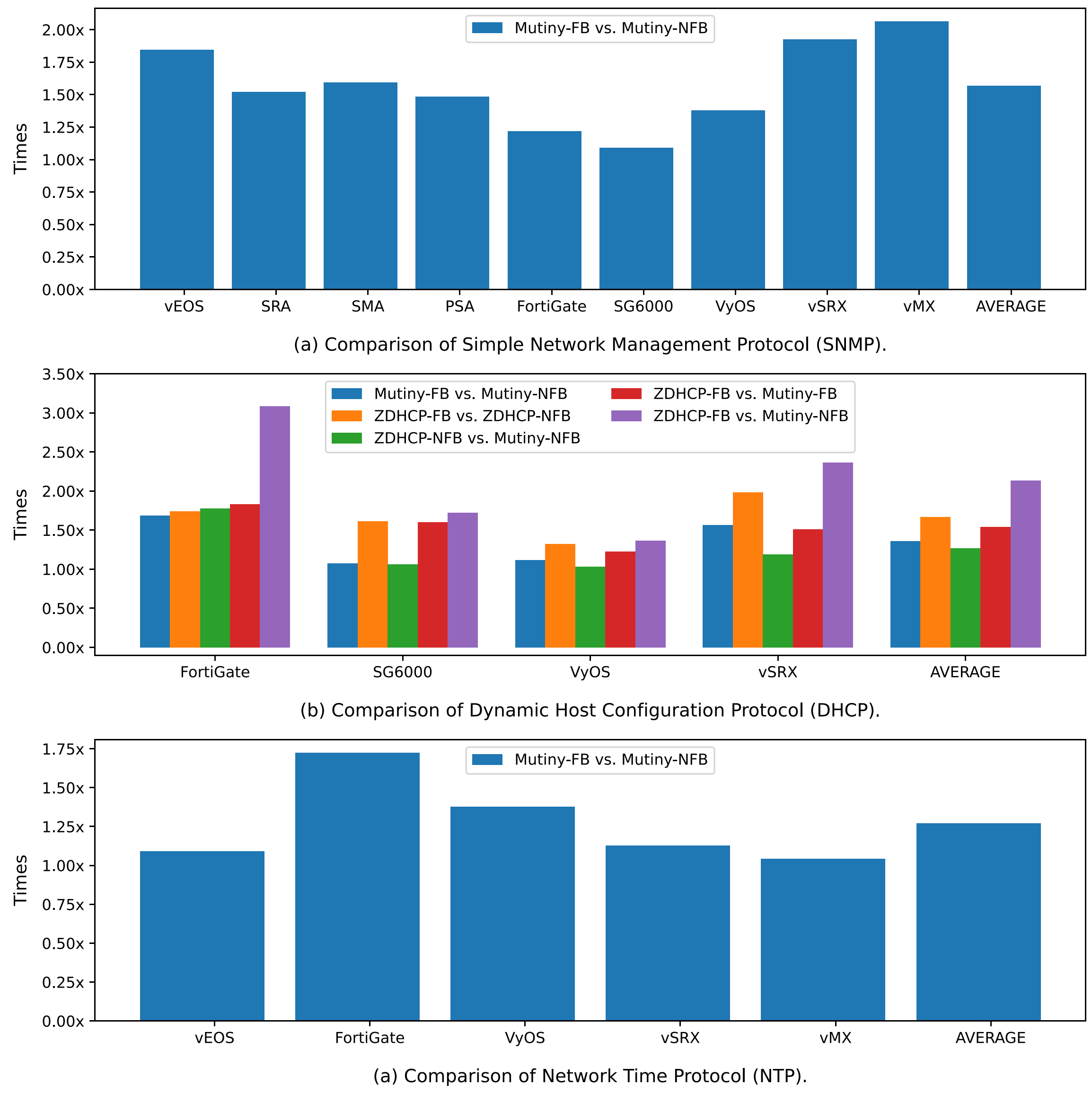


Fig. 6 Edge improvement of SNMP and DHCP protocol

2 *ZDHCP-NFB vs. Mutiny-NFB* Our ZDHCP-NFB also performs better than Mutiny-NFB with an average improvement of 26.72% from 3.48% of VyOS to 77.57% of PSA. This proves the option-focusing mutation can generate the testcase which contains more complex crafted options.
3 *ZDHCP-FB vs. Mutiny-FB* With coverage guidance, the ZDHCP-FB has an average 54.37% improvement than Mutiny-FB. This demonstrates the performance advantage of ZDHCP is further enhanced by the feedback.
4 *ZDHCP-FB vs. Mutiny-NFB* Finally, we compare the ZDHCP-FB with Mutiny-NFB to reveal the improvement by the cooperation of the fuzzing engine and coverage-guidance. The average is 113.67% from 36.88% of VyOS to 208.82% of FortiGate.

The comparison of NTP (Fig. 6c) is similar to SNMP, Mutiny-FB performs an average of 27.25% more edge coverage than Mutiny-NFB from 4.32% of vMX to 72.31% of Fortinet.

Vulnerability discovery

Among the 72 h fuzzing of the 13 fuzzing instances, we totally found numerous crashes and analyzed them manually. Table 5 illustrates the detail information about the

Table 5 Unique crashes statistic

ID	Model	Protocol	Type	Mutiny-NFB	Mutiny-FB	ZDHCP-NFB	ZDHCP-FB
VUL1	FortiGate	DHCP	Integer overflow	✗	✗	24.95 h	9.97 h
VUL2	FortiGate	DHCP	Buffer overflow	✗	33.76 h	✗	1.34 h
VUL3	vEOS	SNMP	Buffer overflow	✗	54.18 h	Not support	Not support

Table 6 Total crashes caught by different monitoring mechanisms

Protocol	SNMP				DHCP							
Fuzzer	Mutiny-NFB		Mutiny-FB		Mutiny-NFB		Mutiny-FB		ZDHCP-NFB		ZDHCP-FB	
Monitor	I[a]	O[a]	I	O	I	O	I	O	I	O	I	O
FortiGate	0	0	0	0	0	0	0	361	40	511	134	1900
vEOS	0	0	0	5	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

[a]I means In-VMM monitoring mechanism

[b]O means Out-of-VMM monitoring mechanism

finding. We have confirmed two unique crashes of DHCP protocol and one of SNMP. After we reported them to the vendors, two were confirmed as 0-Day vulnerability (VUL1 and VUL3) and the other one is already fixed (VUL2). These three vulnerabilities can be both triggered by only one request. Although found from the virtualized network devices, they also affect the physical network devices according to the vendor's response.

VUL1 is firstly found by ZDHCP-NFB in 24.95h and by ZDHCP-FB in 9.97h. This demonstrates the coverage guidance can accelerate the discovering of vulnerability. VUL2 is firstly found by Mutiny-FB in 33.7h and by ZDHCP-FB in 1.34h. Besides showing the advantage of coverage guidance, VUL2 also proves that our customized ZDHCP performs quite better than the general fuzzer Mutiny by saving 96% time.

Compared to DHCP, SNMP has a strict format limitation. Even though with coverage guidance, Mutiny only found one vulnerability about SNMP because it is hard to mutate a legal packet due to the OID encoding. SNMP uses Object Identifier (OID) to manage different objects and the OID parsing accounts for the major part of SNMP process. However, the OID is encoded using Basic Encoding Rules (BER), which is difficult to satisfy with random fuzzing of Mutiny and numerous mutations are dropped during the format check.

Table 6 shows the statistic of the performance of the In-VMM and Out-of-VMM monitoring mechanisms. In fact, this is the initial result without de-duplication, and the result after de-duplication is shown in Table 5. We only list FortiGate and vEOS because they are the two only devices that found vulnerabilities. The two monitoring mechanisms do not catch any crash on the other 7 devices. Firstly, this table depicts that the number of crashes in DHCP is significantly increased with the coverage guidance (551 vs. 2395), which also shows the effectiveness of coverage guidance. Secondly, as described in Subsection Monitoring a Userland Program, if the In-VMM monitoring cannot catch a crash immediately, the Out-of-VMM monitoring can catch it to avoid the false negatives, the "O" columns show the necessity of it.

Overhead of non-intrusively tracing

Evaluating the overhead is not easy. The targets of the mainstream fuzzing benchmarks (Dolan-Gavitt et al. 2016; Hazimeh et al. 2020) are not faced with the daemon process fuzzing, and most of our target devices are unable to execute them. Thus we evaluate them based on our target protocol SNMP, which can always be traversed by a standard program snmpwalk deployed out of the device. By obtaining the execution cost of snmpwalk, we can evaluate the overhead more realistically. We collected the processing time of the snmpd program for communication established by snmpwalk as a benchmark to calculate the overhead from two dimensions.

We hook the CR3 switching function to compute the total time of the only target snmpd process and total operating system time, including all process running. Thus we can calculate the overhead of the only target process and the whole VNF (containing all processes). Figure 7 depicts the statistic of overhead. For each device, the blue bar is the overhead of the single SNMP process and the orange bar is the overhead of the whole VNF.

For the overhead of the edge recording, the whole VNF is from 3.0% (SG6000) to 24.5% (SMA) with an average of 11.7%, this is significantly lower than the target process

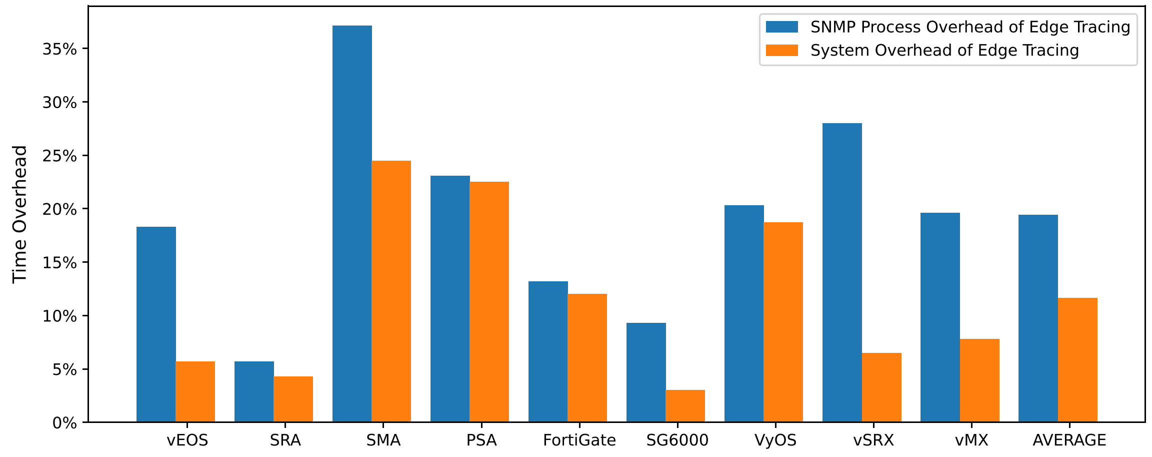


Fig. 7 Overhead of coverage collection

from 9.3% (SG6000) to 37.0% (SMA) with an average 19.4%. This is because customized QEMU only traces the target process and few kernel instructions while other processes' execution, and most kernel instructions are not affected. The overhead focuses on the whole VNF is more related to our real fuzzing scenario, which is acceptable for fuzzing.

Case study

Now we present a detailed case study about the VUL2 to reveal the assistance of coverage guidance. We compare the ZDHCP-NFB and ZDHCP-FB to demonstrate that it is difficult to find VUL2 without feedback.

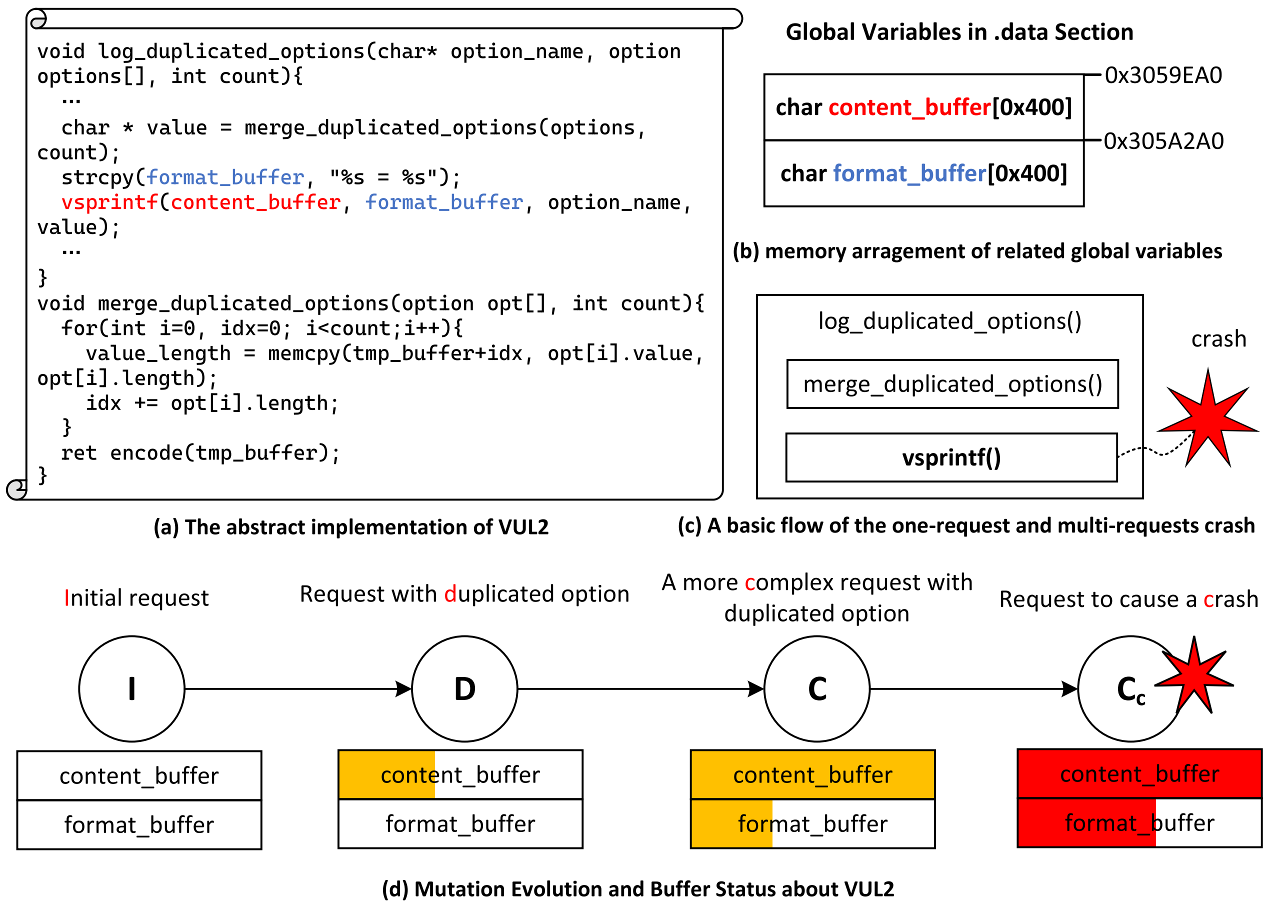
Although VUL2 is a 1-Day issue, it is quite interesting with the root cause of data section overflow. Figure 8 reveals the abstract description about it. As subfigure (b) shows, *content_buffer* and *format_buffer* are two adjacent arrays. The format_buffer stores a format string such as "%s,%d" and content_buffer stores the formatted string which is corresponding to format_buffer. When the DHCP process receives a DHCP request, it parses each option and then calls related functions. Normally, the option types are different from each other but ZDHCP can mutate the request that contains options with the same type (dup-options for short). The DHCP process collects all the same type options and catenates their values as a single value to handle dup-options. Function *merge_duplicated_options()* and *log_duplicated_options()* of Fig. 8a depict the procedure. The root cause is the lack of the length check for the value after catenating. When the value are stored into *content_buffer*, a buffer overflow of data section might be triggered by vsprintf(). During the mutation, the content of the overflow might be changed, which makes the format_string argument of vsprintf() change during vsprintf() running to trigger a crash.

Figure 8d shows the mutation evolution and the colored zones show the buffer status after the request is handled. Based on an initial request I, the first mutation node D contains the options of duplicated type. Handling D would call log_duplicated_options() and the buffers is filled correctly without overflow. Then a more complex request C could be evolved with the more complex option value based on D. This mutation leads to fill more content to content_buffer and might cause overflow to format_buffer. But the overflow might not trigger a crash if no special character overflows the format_buffer. This iteration triggers more edges than D because of the option parsing and catenating. The mutation based on C might gradually change the content and length until a final important request C_c. This mutation contains several format placeholders such as "%s" and more, which leads to the vsprintf() access an illegal address to crash.

Table 7 compares the ratio of different request types of Subfigure (d) about ZDHCP-NFB and ZDHCP-FB. The initial seed contains 3 groups which contains 5, 10 and 20 requests which have the same option to focus on VUL2 detecting. Each fuzzing engine tests at most 1 h and stops when VUL2 is triggered. Although ZDHCP-FB and ZDHCP-NFB can both mutate the D and C types, FB has a quite larger probability (4.9x, 8.1x, 9.3x) to mutate a C type request than NFB.

Discussion

Well-known fuzzers integration We do not integrate well-known gray-box fuzzers into NDFuzz such as AFL++, AFL and more, mainly due to they are not targeted at the network service and the fuzzing workflow are significantly different. Besides, they always rely on static



```
void log_duplicated_options(char* option_name, option
options[], int count){
  …
  char * value = merge_duplicated_options(options,
count);
  strcpy(format_buffer, "%s = %s");
  vsprintf(content_buffer, format_buffer, option_name,
value);
  …
}
void merge_duplicated_options(option opt[], int count){
  for(int i=0, idx=0; i<count;i++){
    value_length = memcpy(tmp_buffer+idx, opt[i].value,
opt[i].length);
    idx += opt[i].length;
  }
  ret encode(tmp_buffer);
}
```

(a) The abstract implementation of VUL2
Global Variables in .data Section
char content_buffer[0x400]
char format_buffer[0x400]
0x3059EA0
0x305A2A0
(b) memory arragement of related global variables
log_duplicated_options()
merge_duplicated_options()
vsprintf()
crash
(c) A basic flow of the one-request and multi-requests crash
Initial request
Request with duplicated option
A more complex request with duplicated option
Request to cause a crash
I
D
C
C_c
content_buffer
format_buffer
(d) Mutation Evolution and Buffer Status about VUL2

Fig. 8 Detailed information about VUL2

Table 7 Ratios and time about request type of VUL2

Type	5 options		10 options		20 options	
	FB	NFB	FB	NFB	FB	NFB
I	51.4%	68.2%	60.7%	66.2%	66.9%	69.9%
D	38.0%	29.7%	25.5%	32.1%	22.9%	29.0%
C	10.6%	2.15%	13.8%	1.7%	10.2%	1.1%
First D	0.0032h	0.0091h	0.0037h	0.00023h	0.0056h	0.0054h
First C	0.01h	0.069h	0.0224h	0.038h	0.0379h	0.0312h
Crash	0.738h	< 1h	0.76h	< 1h	0.56h	< 1h

instrumentation or dynamic binary instrumentation provided by *QEMU user mode* to obtain the coverage, which is inconsistent with the limitation of VNFs which is close-source, well-protected and running supported by *QEMU system mode*. Although AFLNET (Pham et al. 2020) focuses on the network protocol, but it still relies on source code to obtain the coverage. NDFuzz can obtain coverage and this mechanism might be adapted to AFLNET, but the whole fuzzing workflow needs to be heavily refactored especially the implementation of fork-server and the interaction with QEMU system-level emulation (instead of the user-level emulation of AFL-QEMU mode) to fit the VNFs. Therefore this is not a non-trivial workload. We will integrate these famous fuzzers which aim at the open-source software into NDFuzz to apply more excellent fuzzing strategies to VNFs.

Stateful protocol We do not consider fuzzing the stateful protocol because it is another measurement for protocol fuzzing. The VNF is also a pretty data set for this aim, especially some control plane protocols such as BGP, OSPF, and more. NDFuzz could adapt for them in our future work to combine the code coverage and protocol states.

Hardware assistance The implementation of the customized QEMU is based on the TCG instead of KVM, because the KVM solution needs to modify host kernel and is dependent on the Intel-PT to collect the runtime information of the userland process. The TCG scheme has fewer limitations while the KVM scheme has a better performance. This also is a future work to improve NDFuzz.

Architecture adaption We do not find any VNF that works on other architectures except x86 and x86_64, but our method is general and NDFuzz can be adapted to other architectures with minor modification.

Protocol adaption There are many protocols in the virtualized network device while we only implement a customization of DHCP. We will implement more customized fuzzers for specific protocols in the future.

Related work

Fuzzing virtualized network device non-intrusively combines several independent technique of security research. The most related virtual machine introspection (VMI) technique and several hypervisor-assisted greybox fuzzing projects cannot satisify our requirement.

Network function virtualization

For a long time, network equipment has provided network services in the form of proprietary hardware and customized software. This causes the so-called network ossification problem and make the service additions and network upgrades difficult (Li et al. 2018). To overcome these problems, European Telecommunications Standards Institute (ETSI) propose network function virtualization (NFV) to virtualize the network functions that are previously provided by proprietary dedicated hardware (Paper 2012).

As one of the three main components of the NFV framework, Virtualized Network Functions (VNF) are software implementations of network device that can be deployed on a network function virtualization infrastructure (NFVI). The componets of VNF are various such as router, switch, SSL VPN gateways, virus scanners, etc. The famous vendors such as Cisco, Juniper, Fortinet and more are all have their NFV products and provide the virtual products. The software of the VNF is the same as the physical device of the same model.

Actually, the VNF can also be deployed on the COTS computer separately as a virtualized network device with the help of hypervisor such as QEMU/KVM, VMware and Hyper-V. From the view of security research, a virtual network can be tested with the blackbox fuzzing method as a physical network device for the same configuration and functions. However, the virtualized network device can also be treated as a virtual machine. It is obvious that we can obtain more runtime information than the physical network device with the help of customized hypervisor and turn the blackbox fuzzing into coverage-guided greybox fuzzing.

Virtual machine introspection and memory analysis

Over the past decades, virtualization technique has been widely used in numerous fields especially the cloud computing and data centers. The hypervisor (VMM), as the infrastructure of this technique located at the low-level of operating system, pushed system monitoring from traditional in-VM monitoring to out-of-VM monitoring which is known as virtual machine introspection (VMI). Through extracting and reconstructing the guest OS states in the host, VMI takes advantage of the isolation and management of hypervisor and has been widely used in security applications ranging from instrusion detection (Payne et al. 2008), malware analysis (Dolan-Gavitt et al. 2011; Fu and Lin 2012, https://github.com/Cisco-Talos/pyrebox), virtual machine management (Sharif et al. 2009; Srinivasan et al. 2011, https://libvmi.com/) and memory forensics (Fu and Lin 2012, https://github.com/volatilityfoundation/volatility).

The well-known challenge in VMI is semantic gap problem (Dolan-Gavitt et al. 2011), which obstable the extraction of high-level semantic information (e.g., process information of guest) from low-level data such as OS memory dump. A wide solution is related to the kernel data structures obtained from the source code analysis or specific driver execution (Wang et al. 2015, https://libvmi.com/, https://github.com/Cisco-Talos/pyrebox, https://github.com/volatilityfoundation/volatility). Virtuoso (Dolan-Gavitt et al. 2011) and VMST (Fu and Lin 2012) are to directly reuse the legacy binary code of some native inspection programs (e.g., ps, lsmod) to automatically bridge the semantic gap without previous kernel data structure. However, they both bring hugh performance overhead. Hybrid-Bridge (Saberi et al. 2014) combined the offline training from Virtuoso and kernel data redirection from VMST to improvement the performance. It needs both trusted VMs in Fast-Bridge and Slow-Bridge delopy the same OS version as the untrusted VMs.

Memory analysis techniques have a great intersection with VMI techniques in terms of semantic information acquisition. The major difference is that memory analysis focuses more on obtaining semantic information with static analysis such as reverse engineering (Case et al. 2010) data signatures (Lin et al. 2011; Dolan-Gavitt et al. 2009) for kernel semantic information. Firmadyne (Socała and Cohen 2016) provide a straightforword method to emulating-compilation method to predict the struct layout according to the version and config file to obtain the accuracy semantic information. ORIGEN (Feng et al. 2016) combines the data flow analysis and binary search technique to extract some offset-reveal instructions among different versions. However, the signature of a single version and the instruction search might not reliable when handle the significant different versions.

Graybox fuzzing with hypervisor

Graybox fuzzing technique is in middle of blackbox fuzzing (Beizer 1995; Myers et al. 2004) and whitebox fuzzing (Godefroid et al. 2008). Greybox fuzzers leverage the runtime information such as code coverage to guide the fuzzing run. The famous modern fuzzing AFL (http://lcamtuf.coredump.cx/afl/) provide two ways to collect the runtime information: static instrumentation for open-source target or dynamic instrumentation for close-source target (binary). The dynamic instrumentation depend on the QEMU (Bellard 2005) tiny code generator (TCG) mechanism.

The binary fuzzing of AFL works in QEMU user mode which can launch single process. However, it is limited for scenarios related the running operation system such as kernel or emulated firmware fuzzing. TriforceAFL (https://github.com/nccgroup/TriforceAFL) and kAFL (Schumilo et al. 2017) obtain the coverage information by customizing the hypervisor to fuzz the kernel without the source code. REDQUEEN (Aschermann et al. 2019), developed atop of kAFL, can fuzzing the userland program with coverage guidance with QEMU system mode. With the help of hypervisor, it can patch the code related to checksum checks. Firm-AFL (Zheng et al. 2019) which based on Firmadyne (Chen et al. 2016) and DECAF (Henderson et al. 2014) leverage the hypervisor to handle the I/O and syscall support. Fuzzing can take advantage of hypervisor at a little expense of lightweight equipment.

Although graybox fuzzing with hypervisor and VMI both leverage the hypervisor to monitor the runtime information of guest, there are different because of the aim. Different from VMI technique which aims at a macro perspective, fuzzing always foucus on the runtime information such as coverage and process status of a single process or module. Actually, fuzzing with hypervisor only use a small subset of semantic information of VMI, so the semantic gap can be solved with acceptable cost.

Fuzzing embedded devices

Actually, the communication of embedded devices fuzzing is similar to network devices. There have been several works based on networking communication in recent years. IoTFuzzer (Chen et al. 2018) aims at finding the memory corruptions based on hooking the communication logic of related Android apps. SRFuzzer (Zhang et al. 2019) is an automatic blackbox fuzzing framework to discover multi-type vulnerabilities. The shortcomings of black-box fuzzing have been discussed for a long time. Therefore , Snipuzz (Feng et al. 2021) propose a way to infer the running traces by analyzing the response contents to overcome these limitations . Its effectiveness depends on the quality of message snippets which is contingent on how much information could be obtained from devices' responses. Thus it is limited on numerous protocols without this feature (e.g., SNMP , DHCP, NTP and more). Firm-AFL (Zheng et al. 2019) collects the complete trace with AFL-style bitmap with the emulation supported by Firmadyne (Chen et al. 2016). The emulation environment is controllable with a customized kernel, which can not be satisfied by VNFs.

Conclusion

In this paper, we propose a lightweight PGD inference mechanism with differential analysis without any intrusive operations. Leveraging the target PGD values, we present a modular designed framework NDFuzz to fuzz the virtualized network devices with coverage guidance in a non-intrusive way. We have fuzzed nine popular VNFs among seven vendors with NDFuzz and finally found 3 issues during 72h fuzzing with four fuzzing engines.

Abbreviations

NFV: Network function virtualization; VNF: Virtualized network function; OS: Operating system; CLI: Command-line interface; VMM: Virtual machine monitor; VMI: Virtual machine introspection; PGD: Page global directory; TCG: Tiny code generator; DBI: Dynamic binary instrumentation; ARP: Address resolution protocol; NDP: Neighbor discovery protocol; SNMP: Simple network management protocol; OID: Object identifier; DHCP: Dynamic host configuration protocol; KVM: Kernel-based virtual machine.

Acknowledgements

Not applicable.

Authors' contributions

YZ, YYZ and WH designed this research. YZ, NYZ, YYZ, KPJ built this framework and performed experiments. YZ wrote this paper. WY, YYZ, WH, CZ, LBX reviewed and edited the manuscript. All authors read and approved the manuscript.

Funding

This work is supported in part by Chinese National Natural Science Foundation (61802394, U1836209, 62032010). Strategic Priority Research Program of the CAS (XDC02040100).

Availability of data and materials
All public dataset sources are as described in the paper.

Declarations

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. [2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China. [3]Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing, China. [4]Beijing Key Laboratory of Network Security and Protection Technology, Beijing, China. [5]Renmin University of China, Beijing, China.

Received: 24 December 2021 Accepted: 5 March 2022
Published online: 01 November 2022

References
Aschermann C, Schumilo S, Blazytko T, Gawlik R, Holz T (2019) Redqueen: fuzzing with input-to-state correspondence. In: NDSS, vol 19, pp 1–15
Beizer B (1995) Black-box testing: techniques for functional testing of software and systems. Wiley, New York
Bellard F (2005) Qemu, a fast and portable dynamic translator. In: USENIX annual technical conference, FREENIX Track, California, USA, vol 41, p 46
Böhme M, Pham V-T, Roychoudhury A (2017) Coverage-based Greybox fuzzing as Markov chain. IEEE Trans Softw Eng 45(5):489–506
Carbone M, Conover M, Montague B, Lee W (2012) Secure and robust monitoring of virtual machines through guest-assisted introspection. In: International workshop on recent advances in intrusion detection. Springer, pp 22–41
Case A, Marziale L, Richard GG III (2010) Dynamic recreation of kernel data structures for live forensics. Digit Investig 7:32–40
Chen DD, Woo M, Brumley D, Egele M (2016) Towards automated dynamic analysis for Linux-based embedded firmware. In: NDSS, vol 1, pp 1
Chen J, Diao W, Zhao Q, Zuo C, Lin Z, Wang X, Lau WC, Sun M, Yang R, Zhang K (2018) Iotfuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In: NDSS
Cisco-Talos: mutiny fuzzing framework. https://github.com/Cisco-Talos/mutiny-fuzzer
Compare coverage for AFL++ QEMU. https://andreafioraldi.github.io/articles/2019/07/20/aflpp-qemu-compcov.html
Dolan-Gavitt B, Srivastava A, Traynor P, Giffin J (2009) Robust signatures for kernel data structures. In: Proceedings of the 16th ACM conference on computer and communications security, pp 566–577
Dolan-Gavitt B, Leek T, Zhivich M, Giffin J, Lee W (2011) Virtuoso: narrowing the semantic gap in virtual machine introspection. In: 2011 IEEE symposium on security and privacy. IEEE, pp 297–312
Dolan-Gavitt B, Hulin P, Kirda E, Leek T, Mambretti A, Robertson W, Ulrich F, Whelan R (2016) Lava: large-scale automated vulnerability addition. In: 2016 IEEE symposium on security and privacy (SP), pp 110–121. https://doi.org/10.1109/SP.2016.15
Feng Q, Prakash A, Wang M, Carmony C, Yin H (2016) Origen: automatic extraction of offset-revealing instructions for cross-version memory analysis. In: Proceedings of the 11th ACM on Asia conference on computer and communications security, pp 11–22
Feng X, Sun R, Zhu X, Xue M, Wen S, Liu D, Nepal S, Xiang Y (2021) Snipuzz: black-box fuzzing of IoT firmware via message snippet inference. arXiv:2105.05445
Fioraldi A, Maier D, Eißfeldt H, Heuse M (2020) Afl++: combining incremental steps of fuzzing research. In: 14th {USENIX} workshop on offensive technologies ({WOOT} 20)
Fu Y, Lin Z (2012) Space traveling across vm: automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In: 2012 IEEE symposium on security and privacy. IEEE, pp 586–600
Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, Chen Z (2018) Collafl: path sensitive fuzzing. In: 2018 IEEE symposium on security and privacy (SP). IEEE, pp 679–696
Gao Z, Dong W, Chang R, Wang Y (2020) Fw-fuzz: a code coverage-guided fuzzing framework for network protocols on firmware. Concurr Comput Pract Exp
Godefroid P, Levin MY, Molnar DA et al (2008) Automated whitebox fuzz testing. In: NDSS , vol 8, pp 151–166
Han B, Gopalakrishnan V, Ji L, Lee S (2015) Network function virtualization: challenges and opportunities for innovations. IEEE Commun Mag 53(2):90–97. https://doi.org/10.1109/MCOM.2015.7045396
Hazimeh A, Herrera A, Payer M (2020) Magma: a ground-truth fuzzing benchmark. In: Proceedings of the ACM on measurement and analysis of computing systems, vol 4, no 3. https://doi.org/10.1145/3428334
Helin A. Radamsa, a general-purpose fuzzer. https://gitlab.com/akihe/radamsa
Henderson A, Prakash A, Yan LK, Hu X, Wang X, Zhou R, Yin H (2014) Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In: Proceedings of the 2014 international symposium on software testing and analysis, pp 248–258
Jain B, Baig MB, Zhang D, Porter DE, Sion R (2014) Sok: introspections on trust and the semantic gap. In: 2014 IEEE symposium on security and privacy. IEEE, pp 605–620
Li J, Zhao B, Zhang C (2018) Fuzzing: a survey. Cybersecurity 1(1):1–13
LibVMI. https://libvmi.com/
Lin Z, Rhee J, Zhang X, Xu D, Jiang X (2011) Siggraph: brute force scanning of kernel data structure instances using graph-based signatures. In: Ndss
Lyu C, Ji S, Zhang C, Li Y, Lee W-H, Song Y, Beyah R (2019) {MOPT}: optimized mutation scheduling for fuzzers. In: 28th {USENIX} security symposium ({USENIX} security 19), pp 1949–1966
Manès VJM, Han H, Han C, Cha SK, Egele M, Schwartz EJ, Woo M (2019) The art, science, and engineering of fuzzing: a survey. IEEE Trans Softw Eng 47:2312–2331
Mijumbi R, Serrat J, Gorricho J-L, Bouten N, De Turck F, Boutaba R (2016) Network function virtualization: state-of-the-art and research challenges. IEEE Commun Surv Tutor 18(1):236–262. https://doi.org/10.1109/COMST.2015.2477041
Muench M, Stijohann J, Kargl F, Francillon A, Balzarotti D (2018) What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In: Network and distributed system security symposium (NDSS)
Myers GJ, Badgett T, Thomas TM, Sandler C (2004) The art of software testing, vol 2. Wiley, New York
NCCGroup: Project Triforce: run AFL on everything! https://github.com/nccgroup/TriforceAFL
Paper NW (2012) Network functions virtualisation: an introduction, benefits, enablers, challenges & call for action. Issue 1
Payne BD, Carbone M, Sharif M, Lee W (2008) Lares: an architecture for secure active monitoring using virtualization. In: 2008 IEEE symposium on security and privacy (sp 2008). IEEE, pp 233–247
Pham V-T, Böhme M, Roychoudhury A (2020) Aflnet: a greybox fuzzer for network protocols. In: 2020 IEEE 13th international conference on software testing, validation and verification (ICST). IEEE, pp 460–465
PyREBox. https://github.com/Cisco-Talos/pyrebox
Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H (2017) Vuzzer: application-aware evolutionary fuzzing. In: NDSS, vol 17, pp 1–14
Saberi A, Fu Y, Lin Z (2014) Hybrid-bridge: efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In: Proceedings of the 21st annual network and distributed system security symposium (NDSS'14)
Scapy. https://scapy.net/
Schumilo S, Aschermann C, Gawlik R, Schinzel S, Holz T (2017) kafl: hardware-assisted feedback fuzzing for {OS} kernels. In: 26th {USENIX} security symposium ({USENIX} security 17), pp 167–182
Sharif MI, Lee W, Cui W, Lanzi A (2009) Secure in-vm monitoring using hardware virtualization. In: Proceedings of the 16th ACM conference on computer and communications security, pp 477–487
snmpwalk. https://linux.die.net/man/1/snmpwalk
Socała A, Cohen M (2016) Automatic profile generation for live Linux memory analysis. Digit Investig 16:11–24
Srinivasan D, Wang Z, Jiang X, Xu D (2011) Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring. In:

Proceedings of the 18th ACM Conference on computer and communications security, pp 363–374

Volatility framework—volatile memory extraction utility framework. https://github.com/volatilityfoundation/volatility

Wang G, Estrada ZJ, Pham C, Kalbarczyk Z, Iyer RK (2015) Hypervisor introspection: a technique for evading passive virtual machine monitoring. In: 9th {USENIX} workshop on offensive technologies ({WOOT} 15)

Yan LK, Yin H (2012) Droidscope: seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis. In: 21st {USENIX} security symposium ({USENIX} security 12), pp 569–584

Yue T, Wang P, Tang Y, Wang E, Yu B, Lu K, Zhou X (2020) Ecofuzz: adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In: 29th {USENIX} security symposium ({USENIX} security 20), pp 2307–2324

Zalewski M. American fuzzy lop. http://lcamtuf.coredump.cx/afl/

Zhang Y, Huo W, Jian K, Shi J, Lu H, Liu L, Wang C, Sun D, Zhang C, Liu B (2019) Srfuzzer: an automatic fuzzing framework for physical SOHO router devices to discover multi-type vulnerabilities. In: Proceedings of the 35th annual computer security applications conference, pp 544–556

Zheng Y, Davanian A, Yin H, Song C, Zhu H, Sun L (2019) Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In: 28th {USENIX} security symposium ({USENIX} security 19), pp 1099–1114

Publisher's Note