Politecnico di Torino

Master Degree course in Cybersecurity

Master Degree Thesis

A holistic approach for formal adaptive firewall rule management

Supervisors
Prof. Riccardo Sisto
Prof. Fulvio Valenza
Prof. Daniele Bringhenti
Prof. Luca Durante

Candidate
Giovanni de Maria

Academic Year 2024-2025

Summary

Modern network infrastructures require high speeds for multiple services at once and a robust defence against cyber threats. Among these, Denial-of-Service (DoS) attack is a major risk because it consists of flooding systems to make them unavailable. The security provided by conventional firewalls using static rules becomes insufficient when networks experience this type of attack. To address this limitation, this thesis develops a framework that combines verified static configurations with flexible runtime rule distribution.

The system integrates VEREFOO (Verified Refinement and Optimized Orchestrator) which generates optimized and formally verified firewall policies at design time with REDIAL (RulE DIstribution ALgorithm) which operates at runtime to distribute rules between cascaded firewalls. This system is evaluated through testing on various hardware systems and in simulated attack environments with the aim to demonstrate the effectiveness of the framework.

The test-bed replicated an enterprise border network using heterogeneous hardware. Network namespaces were used to separate the functions of routing and filtering. The control-plane automation used SSH-based scripts while experimental data collection occurred on RAM disks to mitigate packet drops when rates reached high levels. The system handled network traffic by running tests that emulated real-world network operations. Attack traffic consisted of ICMP floods of different payload sizes. The experiments included baseline measurements without a firewall and static configuration followed by deployment of VEREFOO–REDIAL in an integrated system to track throughput, packet loss and retransmission.

The baseline measurements showed that unprotected networks became unstable when ICMP floods occurred: small-packet floods broke TCP flows but UDP flows operated at a reduced level. In the final demo, the system integrating the VEREFOO-REDIAL framework proved to be more resistant to attacks. The results show that design-time optimization provides effective configurations when combined with runtime adaptation.

There are three primary limitations: (i) hardware differences that make performance evaluation challenging; (ii) the thesis focuses only on ICMP floods; (iii) the use of iptables-legacy instead of modern packet filtering systems. Future work should evaluate the framework against various attack methods and use contemporary

packet filtering systems and assess the automated policy update capabilities of the REACT-VEREFOO framework. Furthermore, experiments should be conducted on more homogeneous and powerful hardware.

Acknowledgements

I thank all my supervisors. In particular, I am grateful to Prof. Luca Durante for his time and clear guidance. Thanks to my family – especially my parents – my friends and everyone who accompanied me through my university years.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Modern computer network infrastructures operate at high capacity and provide multiple services of different types. The ability to operate in different environments enhances operational flexibility but creates substantial security challenges. One of the most severe is the Denial-of-Service (DoS) attack: it is a serious threat because attackers flood networks and devices by sending large volumes of useless packets. The systems become overloaded under such conditions and then firewalls are required to process traffic in real time.

Firewall rule management is complex. Static configurations provide a first line of defence but they lack adaptability when network traffic patterns change as in the case of a DoS attack. The expanding number of rules in systems leads to system misconfiguration and performance degradation. To improve the defence capabilities of these systems, methods should ensure both correctness and flexibility.

This thesis investigates an integrated solution which unites **VEREFOO** (Verified Refinement and Optimized Orchestrator) [1] with **REDIAL** (RulE DIstribution ALgorithm) [2]. VEREFOO is used at design-time to automatically generate firewall rules that are optimized and formally verified. REDIAL operates at runtime to distribute rules across multiple firewalls which helps balance the load from overloaded nodes and preserves system performance during attacks.

The main goals of this work are:

- **Integration** – VEREFOO and REDIAL form a single pipeline: VEREFOO optimizes the policy at design-time and REDIAL adapts it at runtime.
- **Evaluation** – For this purpose, a realistic test-bed is realized with standard hardware equipment and Linux network namespaces.
- **Demonstration** – The VEREFOO-REDIAL framework effectiveness is validated through a demonstrator.

The thesis is structured as follows. Chapter 2 contains information about the algorithms and tools which are employed. The objectives of the work are presented

in chapter 3. The experimental test-bed is described in chapter 4. The experiments together with their results are presented in chapter 5. Chapter 6 contains conclusions together with suggestions for future work.

Chapter 2

Background

This chapter outlines the main elements that form the basis of the thesis. It first introduces iptables [3] as the underlying firewall technology. Then it presents REDIAL [2], a formally verified rule distribution algorithm for cascaded firewalls and VEREFOO [1], [4], a framework for formally verified configurations. It then describes the measurement tools employed. These elements provide the necessary background for understanding the design, implementation and evaluation of the proposed integrated system.

2.1 iptables

iptables is the primary firewall utility for Linux systems and operates in close integration with the Linux kernel. It inspects each network packet against the configured rule set, applying the **first matching rule**.

A set of rules is grouped into what is known as a **chain** and during packet processing control can be transferred between chains. The most common actions performed by the Linux kernel are **ACCEPT** or **DROP** the packet.

By default, iptables has five independent tables (in practice which tables are present in the configuration depends on the kernel configuration options and the modules installed). Tables are containers of chains.

- **Filter**: the default and most used table, responsible for deciding whether packets are allowed or blocked. It includes `INPUT`, `FORWARD` and `OUTPUT` chains to control traffic to, through and from the system.
- **Nat**: used for address translation when new connections are established (e.g., SNAT, DNAT, masquerading). Includes `PREROUTING`, `INPUT`, `OUTPUT` and `POSTROUTING` chains.
- **Mangle**: handles specialized packet modifications, such as changing Type of Service or marking packets. It has five chains: `PREROUTING`, `INPUT`, `FORWARD`,

`OUTPUT` and `POSTROUTING`.

- **Raw**: used to mark packets for exclusion from connection tracking. Contains only `PREROUTING` and `OUTPUT` chains and is evaluated before other tables.
- **Security**: applies Mandatory Access Control rules after filter rules. Includes `INPUT`, `FORWARD` and `OUTPUT` chains for applying security labels.

The Linux packet processing pipeline is shown in figure 2.1, including all major iptables chains. In this thesis the focus is on the forward path [2].

The default front-end of iptables on current Debian based systems targets the `nf_tables` backend through the `iptables-nft` compatibility layer. The `nf_tables` framework and its translator may apply kernel-side optimizations and encourage stateful constructs beyond simple first-match chains. These could be appropriate in some circumstances, but these behaviours are out of scope for this work and risk altering the effective structure and ordering of the evaluated rule set. To preserve a 1:1 mapping between declared rules and installed rules with deterministic first-match semantics in the classic iptables version, the `iptables` command is pinned to the legacy backend. Since IPv6 is out of scope for the evaluation, `ip6tables` remains unchanged.

The iptables package provides also two standard tools: `iptables-save` [5] and `iptables-restore` [6]. The first is used to export the current firewall configuration into a plain-text format (typically redirected to a file). In contrast, `iptables-restore` reads such a file and re-applies the rule set in a single operation.

This thesis employs iptables because REDIAL relies on rule-based packet filtering with deterministic first-match semantics. Specifically, rules must be redistributed within the `FORWARD` chains of the filter and mangle tables, which inspect packets traversing the firewall from one interface to another (see figure 2.1). This choice matches the operational model of REDIAL and the design assumptions of VEREFOO, both of which concern distributed firewalls. Chains related to traffic directed to or originating from the firewall host itself (`INPUT`, `OUTPUT`) are therefore excluded.

Having justified the use of iptables and the `FORWARD` chain, the discussion now turns to REDIAL.

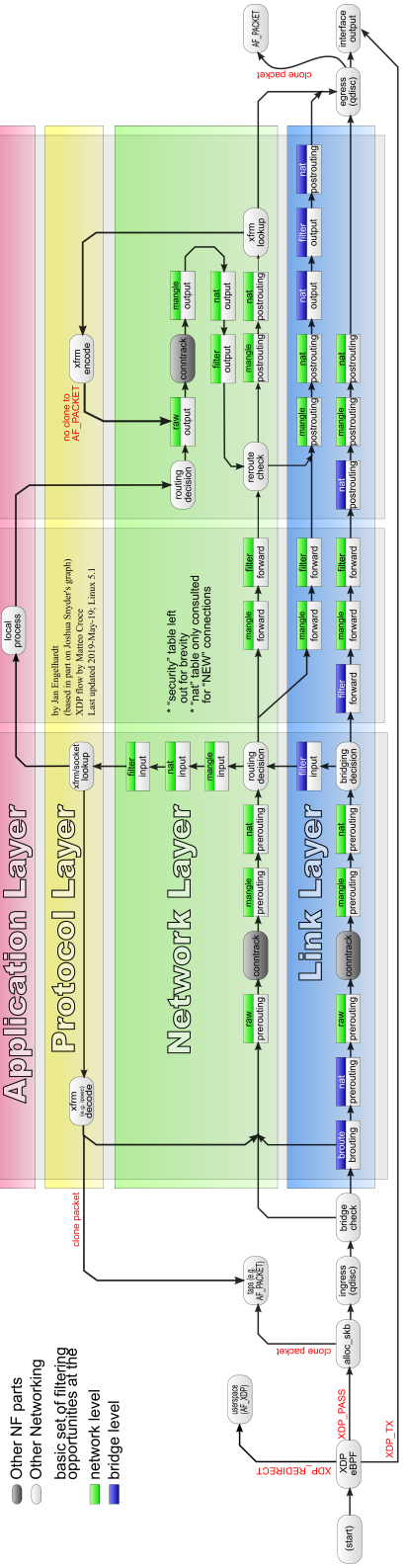


Figure 2.1: The Linux packet processing pipeline (it contains all major iptables chains), source: [7]

2.2 The Rule Distribution Algorithm

REDIAL operates at the operational level and does not require architectural changes. This makes it suitable for industrial networks, since it requires no changes to packet formats or routing paths, and it guarantees that the overall network security policy is maintained after firewalls are reconfigured.

For the purposes of this work, a simplified version of REDIAL [2] is adopted, as defined in algorithm 1.

Figure 2.2 illustrates its operation, while a detailed description of the redistribution process is discussed later in this chapter. It uses a color scheme to represent the relative load of each firewall: red corresponds to nodes experiencing high stress, orange to those involved in the redistribution process and green to nodes functioning under normal conditions.

Operational constraints of the Rule Distribution Algorithm

To apply REDIAL on real Linux firewalls, a set of operational restrictions must be enforced on the iptables configurations. These restrictions mirror those assumed by VEREFOO and remain fully compatible with its model. In particular:

- **Restriction to `FORWARD` path**: only rules placed in the `FORWARD` chains of the mangle and filter tables are considered. Other chains, such as `INPUT` or `OUTPUT`, are excluded since they apply to traffic destined to or originating from the firewall itself and thus fall outside the redistribution scope. As highlighted in figure 2.1, these `FORWARD` chains are part of the forwarding path of Netfilter.
- **No use of connection tracking or marks**: rules that depend on connection state (`conntrack`), marks or related extensions are not supported in redistributed chains.
- **No interface-bound matches**: rules must not contain requirements about input or output interfaces, since their semantics cannot be preserved once moved elsewhere.
- **No SNAT rules for inbound traffic**: source NAT operations in the `POSTROUTING` chain of the `nat` table are not compatible with redistribution and are therefore not allowed.

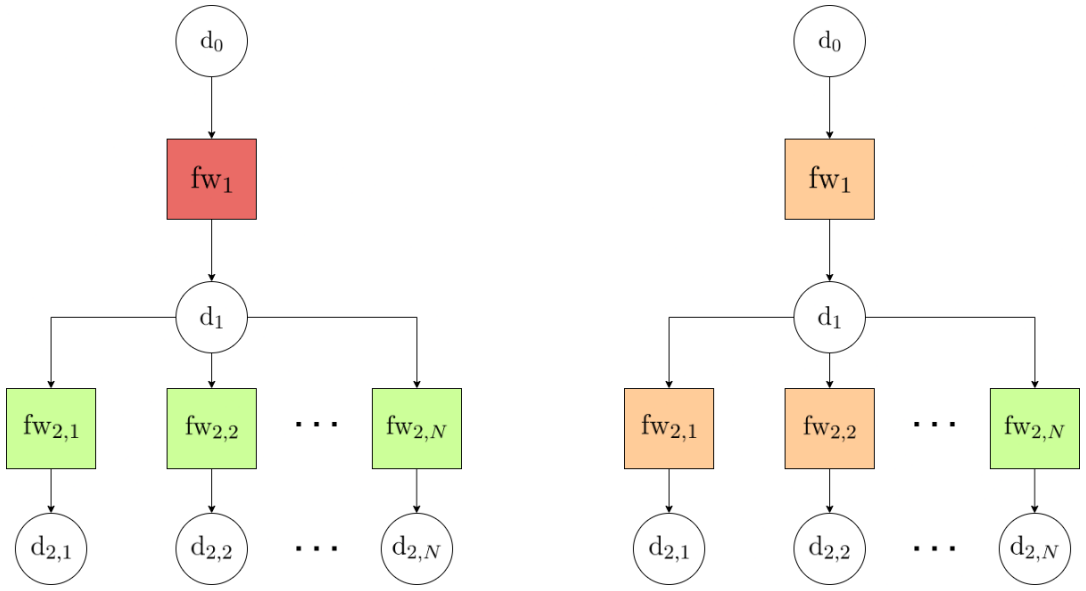
These assumptions define the subset of iptables configurations to which REDIAL can be correctly applied. They are consistent with the reference model used in the formalization and justify using the legacy iptables backend's deterministic first-match behaviour.

Algorithm 1: REDIAL (RulE DIstribution ALgorithm)

Input: $FW = [fw_1^{mf}, fw_1^{ff}]$, D_1^K and D_{K+1}^N from all $d_{2,i}$

Output: Updated chains of the lightened firewall $\widehat{FW} = [\widehat{fw}_1^{mf}, \widehat{fw}_1^{ff}]$
custom chains of downstream firewalls $\widehat{fw}_{2,i}, \forall i \in [1, N]$

Initialize $\widehat{fws}$, $\overline{fws}$ as empty;
Set `flag` $\leftarrow$ `FALSE`;
c ranges over the chains in FW;
i ranges over the rules of fw_1;
$j \in [1, K]$ ranges over the destination sets $d_{2,1}, \ldots, d_{2,K}$;
h ranges over the rules of $\overline{fw}_1$;
$k_1, \ldots, k_K$ range respectively over the rules of $\overline{fw}_{2,1}, \ldots, \overline{fw}_{2,K}$;
All these indices are initially set to 1.
for $c \leftarrow 1$ **to** $|FW|$ **do**
    $fw_1 \leftarrow FW[c]$;
    while $i \leq |fw_1|$ **do**
        while $j \leq K$ **do**
            if $fw_1.r_i.C[d_{ip}] \cap d_{2,j} \neq \emptyset$ **then**
                Add $fw_1.r_i$ to $\overline{fw}_{2,j}$ as r_{k_j};
                $k_j \leftarrow k_j + 1$;
                if *flag* = *FALSE* **then**
                    `flag` $\leftarrow$ `TRUE`;
                    Add $\langle \text{any}, D_1^K, \text{any}, \text{any}, \text{any}, \text{accept} \rangle$ to $\overline{fw}_1$ as r_h;
                    $h \leftarrow h + 1$;
            $j \leftarrow j + 1$;
        if $fw_1.r_i.C[d_{ip}] \cap D_{K+1}^N \neq \emptyset$ **then**
            Add $fw_1.r_i$ to $\overline{fw}_1$ as r_h;
            $h \leftarrow h + 1$;
        $i \leftarrow i + 1$;
    $\widehat{FW}[c] \leftarrow \overline{fw}_1$;
for $i \leftarrow 1$ **to** K **do**
    $\widehat{fw}_{2,i}.r_1 \leftarrow \langle \text{any} \backslash d_0, \text{any}, \text{any}, \text{any}, \text{any}, \text{return} \rangle$;
    for $j \leftarrow 2$ **to** $|\overline{fw}_{2,i}| + 1$ **do**
        $\widehat{fw}_{2,i,r_j} \leftarrow \overline{fw}_{2,i,r_{j-1}}$;



(a) Initial configuration before applying REDIAL

(b) Final configuration after applying REDIAL

Figure 2.2: Example of rule redistribution using REDIAL

Detailed description of the Rule Distribution Algorithm

Algorithm 1 is defined within a hierarchical topology which is formally termed **reference stream topology** (RST) of which figure 2.2 represents a specific instance. The notation employed throughout is summarised in table 2.1, which is introduced here at the outset for clarity. The flows originate from a designated source domain d_0 and propagate through one or more intermediate firewalls before reaching the destination domains $d_{2,i}$. The RST address sets are assumed to be pairwise disjoint and non-interleaved: $d_0 \cap d_{2,i} = \varnothing$ and $d_{2,i} \cap d_{2,j} = \varnothing$ for $i \neq j$.

For each routing device d_1, the address set equals the union of its children, that is $d_1 = \bigcup_i d_{2,i}$. The upstream firewall $FW = [fw_1^{mf}, fw_1^{ff}]$ is composed of the `FORWARD` chain of the mangle table fw_1^{mf} and the `FORWARD` chain of the filter table fw_1^{ff}. Each downstream domain is associated with a set of IP addresses $d_{2,i}$, protected by its corresponding firewall $fw_{2,i}$. When an overload condition is detected at fw_1, the rules that define the handling of traffic from d_0 to a subset of domains are replicated (transferred or duplicated as needed) to the downstream firewalls that directly protect those domains.

Alongside this replication, fw_1 is extended with a single `ACCEPT` rule for D_1^K, preventing any further inspection of that traffic. Packets from d_0 to the delegated domains bypass inspection at fw_1; instead filtering is handled by the appropriate downstream firewalls. Rules for all other destinations D_{K+1}^N remain in fw_1.

Formally, each rule in fw_1 is examined in turn. If the destination matches one

of the domains $d_{2,j}$, the rule is inserted into the configuration of the respective downstream firewall $fw_{2,j}$. Once the first match occurs, the upstream firewall is extended with the global `ACCEPT` rule for D_1^K. Rules applying to destinations outside this set are retained upstream. To ensure correctness, each downstream firewall is also provided with a preliminary rule whose source condition corresponds to any $\setminus d_0$, thereby ensuring that only packets originating from the root domain are considered and preventing interference with unrelated traffic. The resulting configurations

$$\widehat{FW} = [\widehat{fw}_1^{mf}, \widehat{fw}_1^{ff}], \quad \widehat{fw}_{2,1}, \ldots, \widehat{fw}_{2,K}$$

remain semantically equivalent to the original rule set, preserving the same acceptance and rejection behaviour for every packet while distributing the filtering workload more evenly. The formal proof can be found in [2].

Table 2.1: Notation used in REDIAL

Symbol	Meaning
$FW = [fw_1^{mf},\ fw_1^{ff}]$	Configuration of the upstream firewall considered by REDIAL, consisting of the *mangle* table fw_1^{mf} and the *filter* table fw_1^{ff}.
$d_{2,i}$	Set of destination IP addresses belonging to the i-th downstream domain protected by firewall $fw_{2,i}$.
$D_1^K = \bigcup_{i=1}^{K} d_{2,i}$	Union of destination sets for the K downstream domains that participate in redistribution.
$D_{K+1}^N = \bigcup_{i=K+1}^{N} d_{2,i}$	Union of destination sets for downstream domains not affected by redistribution.
$\widehat{FW} = [\widehat{fw}_1^{mf},\ \widehat{fw}_1^{ff}]$	"Lightened" upstream configuration obtained by removing rules targeting D_1^K; it retains only rules applicable to D_{K+1}^N.
$fw_{2,i}$	Custom rule chain(s) installed on downstream firewall $fw_{2,i}$ to enforce rules matching $d_{2,i}$.
$fw_1.r_i$	The i-th rule of the upstream firewall fw_1.
$fw_1.r_i.C$	Match condition of rule r_i (e.g., source/destination addresses, ports, protocol).
$fw_1.r_i.C[d_{ip}]$	Destination-address component of the match condition for rule r_i.

Continues on the next page

Symbol	Meaning
$\langle$ `src`, `dst`, `sport`, `dport`, `proto`, `action` $\rangle$	Tuple representation of a firewall rule, with fields for source/destination addresses, source/destination ports, protocol and action (e.g., `ACCEPT`, `DROP`).
any $\setminus d_0$	Set of all source addresses excluding those in domain d_0.
$FW[c]$	The c-th table in the upstream firewall configuration (either fw_1^{mf} or fw_1^{ff}).

Clarifying example

The following example illustrates how REDIAL works on a simple network shown in figure 2.3. The source domain is defined as $d_0 = 10.0.0.0/8$. Table 2.2 shows the original `FORWARD` chain of fw_1, which contains explicit rules for each subnet ($d_{2,1} = 192.168.1.0/24$, $d_{2,2} = 192.168.2.0/24$, $d_{2,3} = 192.168.3.0/24$) and one broad rule on TCP/443 covering the whole $d_{2,1} \cup d_{2,2} \cup d_{2,3} = 192.168.0.0/16$ range. In this initial state, the upstream firewall must evaluate both fine-grained rules (e.g., specific hosts or ports) and generic rules (such as HTTPS to the entire /16 network).

After applying REDIAL, table 2.3 shows three distinct outcomes. In the **delegation case**, fw_1 replaces the $d_{2,1}$ and $d_{2,2}$ rules with pass-through entries. These rules match any traffic for those subnets and forward it to the respective downstream firewalls, which then enforce the detailed policies. This shows how REDIAL adds a delegation rule that stops inspection upstream and avoids repeating the same checks.

In the **duplication case**, the HTTPS rule on TCP/443 is marked as duplicated, since its address space overlaps both delegated ($D_1^2 = d_{2,1} \cup d_{2,2}$) and non-delegated ($D_3^3 = d_{2,3}$) domains. In this case, the rule is kept at the upstream level and replicated downstream to maintain consistent policy application across both delegated and non-delegated domains.

Finally, in the **retention case**, the rule for $d_{2,3}$ remains on fw_1 because this subnet is not delegated; the redistribution is not needed and the rule must be retained upstream.

After redistribution, as shown in table 2.3, the rules for $d_{2,1}$ and $d_{2,2}$ are replaced by generic delegation rules: packets matching those destinations are no longer inspected upstream, but forwarded to the respective downstream firewalls. The mixed HTTPS (Proto: TCP, Port: 443) rule is marked as duplicated, since it matches both delegated $D_1^2 = d_{2,1} \cup d_{2,2}$ and non-delegated $D_3^3 = d_{2,3}$ sets. Rules for $d_{2,3}$ remain at the upstream device because it belongs to the non-delegated set. The resulting downstream configurations are reported in tables 2.4 and 2.5.

Table 2.2: Upstream fw_1 rules before REDIAL (filter/`FORWARD`)

#	Source	Destination	Proto	Port(s)	Action
1	10.0.0.1	192.168.2.1	TCP	25	ACCEPT
2	10.0.0.0/8	192.168.1.0/24	TCP	22	ACCEPT
3	10.0.0.0/8	192.168.2.0/24	UDP	53	ACCEPT
4	10.0.0.0/8	192.168.3.0/24	ICMP	echo-req	ACCEPT
5	10.0.0.0/8	192.168.0.0/16	TCP	443	ACCEPT
6	any	any	any	–	DROP

Table 2.3: Upstream fw_1 rules after REDIAL (filter/`FORWARD`)

#	Source	Destination	Proto	Port(s)	Action
1	10.0.0.0/8	192.168.1.0/24	any	–	ACCEPT (delegated)
2	10.0.0.0/8	192.168.2.0/24	any	–	ACCEPT (delegated)
3	10.0.0.0/8	192.168.3.0/24	ICMP	echo-req	ACCEPT (retained)
4	10.0.0.0/8	192.168.0.0/16	TCP	443	ACCEPT (duplicated)
5	any	any	any	–	DROP

Table 2.4: Downstream $fw_{2,1}$ rules after REDIAL (filter/`FORWARD`)

#	Source	Destination	Proto	Port(s)	Action
1	10.0.0.0/8	192.168.1.0/24	UDP	53	ACCEPT
2	10.0.0.0/8	192.168.0.0/16	TCP	443	ACCEPT
3	any	any	any	–	DROP

Table 2.5: Downstream $fw_{2,2}$ rules after REDIAL (filter/`FORWARD`)

#	Source	Destination	Proto	Port(s)	Action
1	10.0.0.1	192.168.2.1	TCP	25	ACCEPT
2	10.0.0.0/8	192.168.2.0/24	TCP	22	ACCEPT
3	10.0.0.0/8	192.168.0.0/16	TCP	443	ACCEPT
4	any	any	any	–	DROP

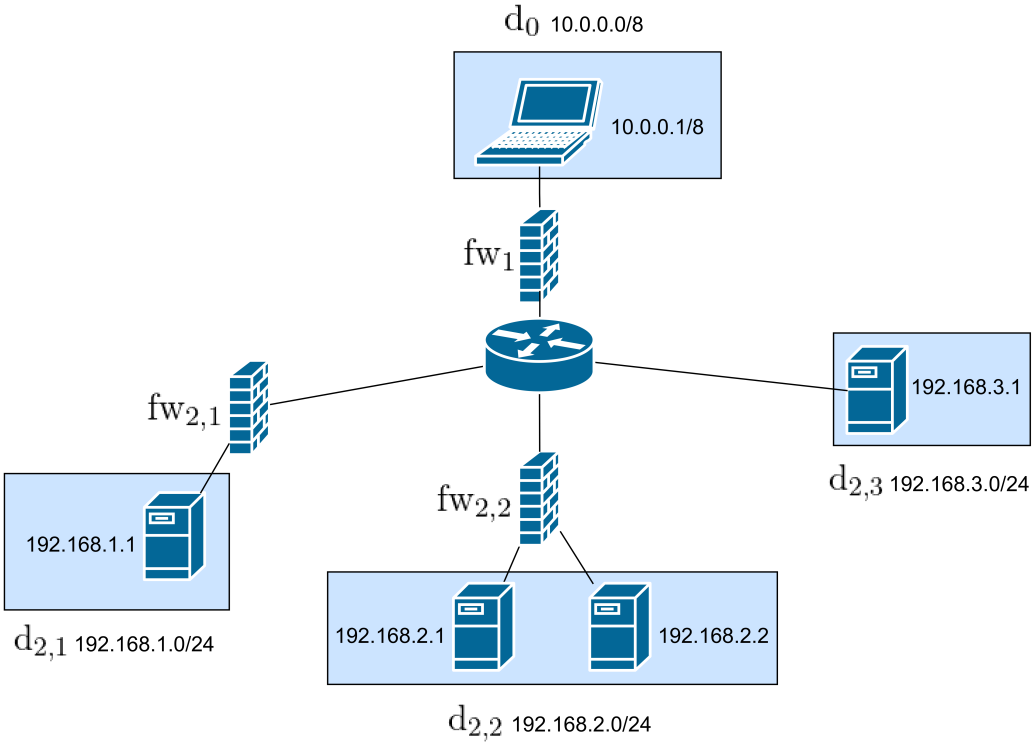


Figure 2.3: Topology for the REDIAL clarifying example

2.3 VEREFOO

The VEREFOO framework is used for configuring distributed firewalls. It takes high-level security requirements (intents) and computes an optimal firewall deployment and rule set that satisfies all policies. VEREFOO combines automation, formal verification and optimization to produce minimal, correct configurations. The firewall configuration problem is expressed as a MaxSMT (Maximum Satisfiability Modulo Theories) optimization (more specifically, VEREFOO relies on the Z3 theorem prover [8], [9]), ensuring the minimal number of rules while meeting all reachability/isolation requirements.

Existing literature [10] explains that manual firewall configuration is error-prone and does not scale with network complexity; however automating this process through tools like VEREFOO improves security and reduces rule complexity.

Bringhenti et al. describe VEREFOO's approach to optimal firewall orchestration in virtualized networks [4] and a recent journal paper explains the automated firewall configuration using formal methods [11].

VEREFOO models the network as a Service Graph (it abstracts the topology

of the infrastructure and identifies the allocation places where firewalls may be deployed) and accepts a set of Network Security Requirements that include both reachability and isolation constraints between endpoints.

The constraints relevant in the context of VEREFOO are the following:

- **Hard constraints:** traffic flows that must be allowed or denied.
- **Soft constraints:** minimizing the number of deployed firewalls and reducing the complexity of the rule sets.

Z3 enforces all hard constraints and minimizes a weighted objective over the soft constraints.

VEREFOO can apply two traffic abstraction models:

- **Atomic Predicates (AP)**: This model partitions the space of traffic predicates (e.g., "source IP in X and destination port equals Y") into a finite set of non-overlapping classes.
- **Maximal Flows (MF)**: In contrast, in this other model, similar traffic flows are treated uniformly.

When a valid solution is found, VEREFOO provides the translation of the model into deployable firewall rules. This includes:

- **Firewall placements:** it is specified which nodes in the Service Graph (also called “Allocation Places”) host firewall instances.
- **Rule logic generation:** the allow or deny policies are set depending on the policies specified as input and the default action is also included.

These outputs are first serialized into a high-level XML format and then translated into low-level platform-specific configuration scripts (iptables, Open vSwitch or IpFirewall). The outputs of VEREFOO are then deployed on iptables-based firewalls. The workflow of VEREFOO is illustrated in figure 2.4.

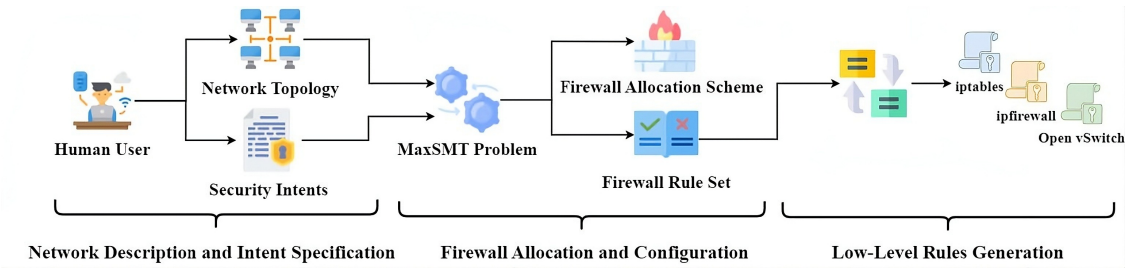


Figure 2.4: VEREFOO Architecture, source: [11]

2.4 Performance measurement techniques and tools

This section outlines the methods and tools used to evaluate firewall performance, ranging from standard benchmarking utilities to custom traffic generators and packet-level inspection frameworks. Together, these provide both aggregate throughput measurements and detailed timing analysis, which were subsequently processed in Python notebooks using standard statistical and visualization libraries (**NumPy**, **Pandas**, **Matplotlib** and **tshark**).

iPerf3

iPerf3 [12], [13] is a widely used tool for network performance assessment tasks. In this work, iPerf3 was used in the initial benchmark phase to establish baseline performance in the absence of firewall filtering. It operates in a standard client-server model: the server must be started on one host and the client connects to it to initiate the test. Both TCP and UDP protocols are supported: in TCP mode, the bandwidth availability is assessed under congestion control and flow control mechanisms, while in UDP mode, fixed data rates are set for measuring packet loss and throughput.

iPerf3 is useful for measuring throughput, but its bursty traffic patterns make it less suitable for precise packet-level analysis; this limitation motivated the development of custom Python scripts able to reproduce controlled request–response sequences at high packet rates.

Hardware timestamping (IEEE 1588)

The experimental campaign relies on hardware timestamping with a **PTP-capable NIC** (IEEE 1588 [14]) by using `tcpdump` [15] and `tshark` [16]. Both tools sit atop the **libpcap** [17] library. A schematic overview is shown in figure 2.5.

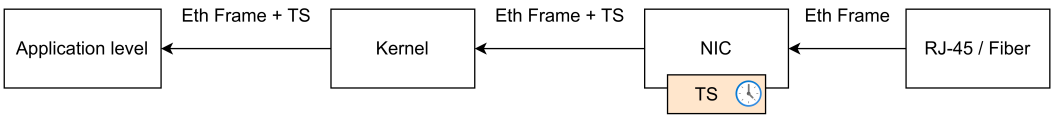


Figure 2.5: Schematic representation of Hardware Timestamping using Raw PHC

Linux systems expose PTP-capable NICs through PHC (PTP Hardware Clock) devices that can be accessed via `/dev/ptpX`. The PHC must be synchronized with the system clock through `phc2sys` and NTP/chrony services need to be disabled during measurements to prevent clock conflicts. The Linux timestamping API transfers NIC-generated timestamps from packets to user space using `sk_buff` and other data [18]. The Intel X550 NIC enables hardware timestamping operations for

both received and transmitted packets. Listing 2.1 shows the output of the `ethtool -T` command; the lines 5,7,10 and 11 confirm the timestamping capabilities of the Intel X550 NIC.

```
1  guest@ThinkStation-P360-main-rtr:~$ ethtool -T enp1s0f1
2
3  Time stamping parameters for enp1s0f1:
4  Capabilities:
5      hardware-transmit
6      software-transmit
7      hardware-receive
8      software-receive
9      software-system-clock
10     hardware-raw-clock
11 PTP Hardware Clock: 1
12 Hardware Transmit Timestamp Modes:
13     off
14     on
15 Hardware Receive Filter Modes:
16     none
17     all
```

Listing 2.1: Timestamping parameters with `ethtool`

The field `PTP Hardware Clock: 1` indicates that the `enp1s0f1` interface is associated with the device `/dev/ptp1` (it exposes the NIC's internal hardware clock).

At the kernel level, Linux provides packet timestamping through the socket option `SO_TIMESTAMPING`. Once enabled, the kernel attaches timestamp metadata to packets and makes it available to user space as ancillary data obtained with `recvmsg()`. For incoming packets, the timestamps are delivered together with the payload in user space, whereas for outgoing packets, they are queued in the kernel's error queue and retrieved from user space using the `MSG_ERRQUEUE` flag [19], [20].

`hping3` as an ICMP flood generator

`hping3` [21], [22] functions as a packet generation tool for network testing and security auditing. The primary use of the tool involves crafting packets to test firewalls and intrusion detection systems and it can generate high-rate ICMP traffic, which is useful for experimental studies of network resistance to flooding attacks.

The ICMP flood attack consists of sending an unending sequence of ICMP Echo Request packets to a specific target network or host. Each packet requires minimal per-packet processing cost along with minimal bandwidth from the victim system. The combination of many such packets creates network link saturation or system resource exhaustion.

The tool offers precise control by allowing the adjustment of several parameters (e.g., packet size, transmission speed and source address). It uses raw sockets, thereby bypassing the standard protocol stack. This feature of the tool enables higher transmission speeds than the built-in `ping` utility and allows delays between packets with a resolution in the order of microseconds.

Chapter 3

Thesis objectives

The core purpose of this thesis is to design and evaluate an integrated firewall management system that combines static, formally verified configuration with dynamic rule redistribution. Modern network infrastructures employing distributed firewalls face a dual challenge: the need to establish an initial configuration that is both efficient and compliant with security requirements and the necessity to adapt automatically to unexpected traffic patterns without manual intervention. This work combines VEREFOO (Verified Refinement and Optimized Orchestrator), a framework for formal rule generation and optimization, with REDIAL (RulE DIstribution ALgorithm), which operates at runtime to redistribute filtering rules and balance processing loads. VEREFOO operates during the modelling phase. It uses a MaxSMT formulation to translate high-level security policies into optimal rule sets, which are then processed by the Z3 solver. All hard constraints are preserved, while rule complexity is minimized. The resulting policy is translated into platform-specific commands—in this case, iptables rules targeting the `FORWARD` chain. In contrast, REDIAL functions during the runtime phase. When REDIAL is triggered by an abnormal event it transfers the load to downstream devices located closest to the target network segments. This redistribution reduces per-packet processing overhead, helping to maintain responsiveness and throughput during high-load conditions, while preserving the original security semantics.

3.1 Integration objective

The first objective is to integrate VEREFOO's design-time optimization with REDIAL's runtime flexibility in a unified workflow:

1. During the modelling phase, VEREFOO computes an optimized configuration from the stated policy intents.

2. During the runtime phase, REDIAL performs the redistribution of rules to address traffic imbalances without requiring a complete policy re-computation.

This integration safeguards the formal assurance provided at design-time and adds the flexibility needed to manage network variability in real environments. Figure 3.1 shows the operational workflow of the integrated VEREFOO–REDIAL system. VEREFOO compiles initial security policies into an optimized rule set. When an attack occurs, REDIAL performs runtime redistribution of rules to mitigate the effects and updates are fed back into the policy configuration.

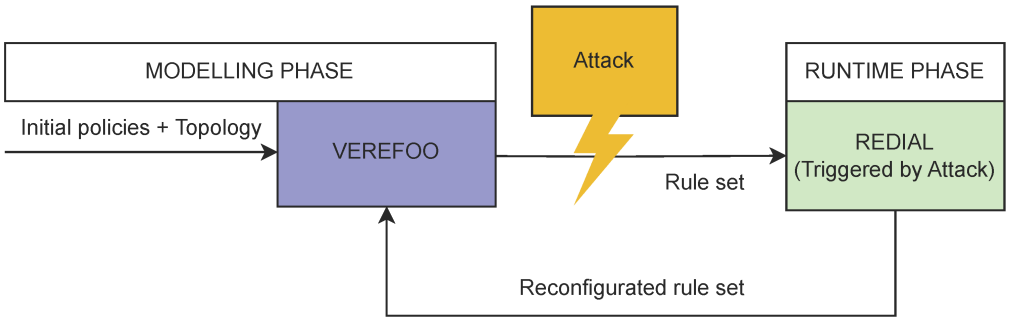


Figure 3.1: Operational workflow of the integrated VEREFOO–REDIAL system

3.2 Performance evaluation objective

The second objective is to assess the behaviour of the framework under realistic conditions – including both normal operation (without firewalls) and attack scenarios. The testing environment emulates a small enterprise border network featuring:

- Legitimate external client traffic to public services, in compliance with established policies.
- The attacks are ICMP floods that exhaust the servers' processing capacity.

The evaluation will focus on:

- **Throughput** of legitimate connections.
- **Packet loss** across the network with attention to legitimate flows.

3.3 Mitigation demonstration objective

This objective is to demonstrate the practical advantages of runtime reconfiguration. The purpose is to show that REDIAL can reduce load on the most stressed firewalls. The thesis aims to obtain both policy correctness (achieved at design-time) and the resilience of the systems (provided by real-time adaptation) to deliver a network defence system that is both robust and responsive.

Chapter 4

Experimental test-bed setup

This chapter describes the environment used for the experiments. The test-bed emulates the perimeter of an enterprise network with Internet-facing services, internal server domains and distributed security elements.

The **scenario model** (logical zones and roles) is treated separately from the **implementation** (physical devices and namespaces in the laboratory).

The implementation relies on mainstream hardware and Linux operating systems. Isolation is provided by Linux network namespaces; reproducibility is supported by fixed addressing, scripted configuration and packet capture on wired links.

A separate management network provides remote access and out-of-band control, ensuring that experimental traffic remains isolated.

4.1 Scenario model

The environment mirrors an authentic enterprise perimeter, which includes Internet-facing services including internal server segments and distributed filtering elements. The following design goals guided the model:

- **Realism**: the model includes external clients, attackers, internal servers and border firewalls as found in real deployments.
- **Isolation**: functional roles (routing, filtering and endpoints) are separated.
- **Reproducibility**: the addressing plans and roles are fixed, so the experiments can be repeated.

Logical segmentation

As shown in figure 4.1, the modelled environment is divided into four areas:

- **Internal server zone**: application servers, which are shielded by two internal firewalls.
- **Secondary internal zone**: internal hosts reachable only from within the corporate network.
- **External zone**: public Internet segment outside the network, one legitimate client and two distinct attackers.
- **Border router**: a single element implementing both routing and packet filtering.

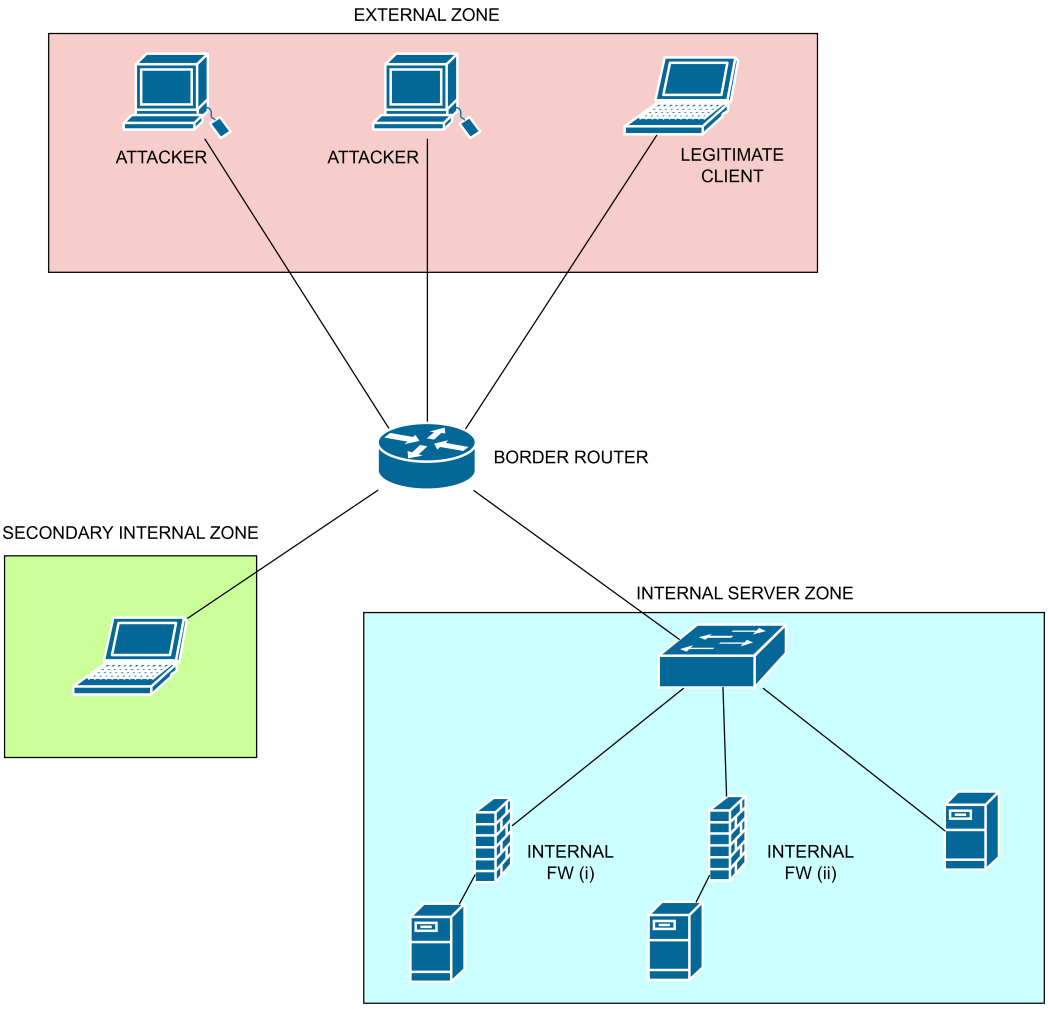


Figure 4.1: Abstract enterprise perimeter model used in the experiments

4.2 Implementation

Hardware inventory and Operating Systems

The physical devices range from Raspberry Pis to workstation-grade machines:

- Three *Raspberry Pi 5 Model B* boards, each equipped with a 1 Gbps Ethernet interface, running Ubuntu 25.04.
- Two *Raspberry Pi 3 Model B* with a 100 Mbps Ethernet interface and running Raspbian 12 Bookworm.
- One *Raspberry Pi 2 Model B* with a 100 Mbps Ethernet interface, running Raspbian 12 Bookworm (32-bit version).
- One *Lenovo ThinkStation P5* (model 30GA000NIX) as the attack generator, equipped with two Edimax EN-9260TX-E (1 Gbps) NICs and running Ubuntu 24.04.2 LTS.
- One *Lenovo ThinkStation P360 Tower* (model 30FMS1EK00) with two Intel X550 NICs (two ports each, up to 5 Gbps per port) and one Edimax EN-9260TX-E (1 Gbps) NIC, running Ubuntu 24.04.2 LTS.

Each node's role varies depending on the experiment. All experimental links use **wired Ethernet** (Wi-Fi is not used) to maximize the stability and precision of the measurements. Wi-Fi is used only on the parallel control network; see section 4.3.

Model-to-implementation mapping

Each modelled role is realised in the laboratory by assigning it to a namespace on a specific physical device. Figure 4.3 shows the implementation topology, with coloured boxes indicating devices that host multiple logical functions (separate Linux namespaces) and unboxed icons corresponding to single-namespace hosts. Interface IP addresses and namespace names are shown directly in the figure, while table 4.1 summarises how these roles are implemented. Figure 4.2 depicts the physical test-bed configuration.

Table 4.1: Mapping between abstract model roles and physical implementation.

Physical Device	Namespace(s)	Role(s) in Model	IP(s)
ThinkStation P360	`fw`, `rtr`	Border router: `fw` for packet filtering, `rtr` for routing	10.0.2.254, 10.0.3.254, 9.0.0.254, 9.0.1.254, 9.0.2.254
ThinkStation P5	`att1`, `att2`	External attackers (two distinct namespaces emulate separate sources)	9.0.0.1, 9.0.2.1
Raspberry Pi 5 (i)	`fw`, `edp`	Internal firewall (i) in `fw`; application server (i) in `edp`	10.0.3.1, 10.0.3.11
Raspberry Pi 5 (ii)	`fw`, `edp`	Internal firewall (ii) in `fw`; application server (ii) in `edp`	10.0.3.2, 10.0.3.21
Raspberry Pi 3	`edp`	Unprotected host in the server zone, emulated in `edp`	10.0.3.3
Raspberry Pi 2	`edp`	Secondary internal hosts emulated in `edp`	10.0.2.1–3

Address blocks are chosen to disambiguate functional roles across the test-bed:

- **10.0.3.0/24**: internal server zone and internal firewalls. A simple Layer-2 switch interconnects the two internal firewalls and the unprotected host (10.0.3.3). The application servers, 10.0.3.11 and 10.0.3.21, are instead protected by firewalls and are not directly attached to the switch.
- **10.0.2.0/24**: secondary internal network with multiple logical IPs assigned to the same interface.
- **9.0.0.0/24**, **9.0.1.0/24**, **9.0.2.0/24**: external Internet zones housing attackers and the legitimate client.

Interface names are those assigned by the operating system.

Linux network namespaces [23] permit multiple logical nodes in the same physical device and allow reproducibility. As explained in section 2.1, the main purpose of this design choice is to meet the functional requirements of VEREFOO and REDIAL which both operate on the `FORWARD` chain. The use of separate namespaces allows to have better control plane and test network separation, which results in improved isolation and management during experiments.

Namespaces are interconnected at Layer 2 through Virtual Ethernet pairs (`veth`). One end is assigned to the `fw` namespace and the other one to `rtr` or to `edp` depending on the device that is considered.

The namespaces labelled `fw`, corresponding to the two internal firewalls, are implemented as Linux bridges that attach the physical NIC and the virtual Ethernet pair; the namespace IP address is therefore assigned to the bridge device.

Figure 4.2: Physical laboratory setup

To make the logical split explicit, the full setup script used on the central router is reported in appendix A.1. The script requires root privileges to run and accepts commands for creating or clearing the network while it establishes the `fw` and `rtr` namespaces through a dedicated `veth` /30 network and moves physical NICs to their respective namespaces before assigning IPv4 addresses and enabling L3 forwarding. The NIC names (e.g., `enp2s0`, `enp1s0f0`) are specific to the ThinkStation P360 hardware and need to be adjusted on other hosts. This listing is provided solely as an example; analogous scripts (not shown) configure the attacker, client and server nodes with the same pattern of namespace creation, addressing and routing.

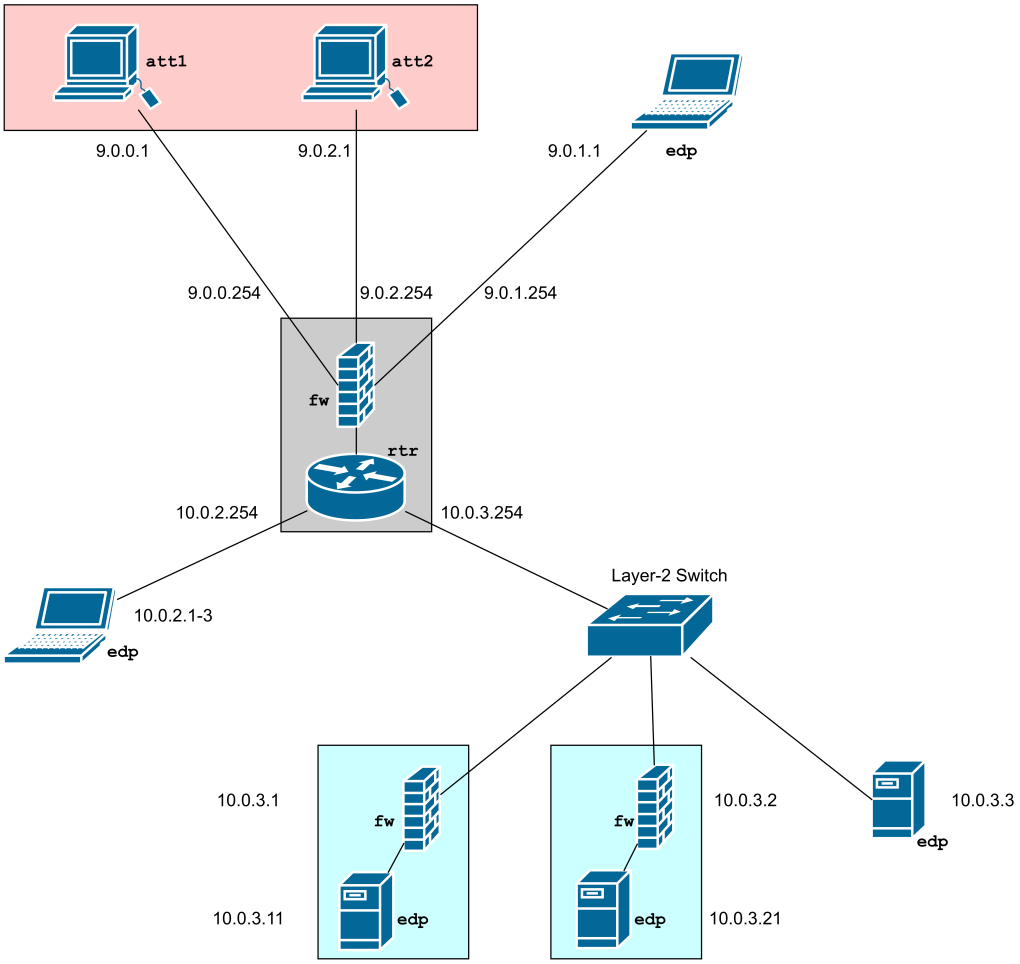


Figure 4.3: Implementation topology of the experimental test-bed

4.3 Out-of-band control network and remote access

To guarantee remote access for configuration and monitoring without interfering with experimental traffic, a dedicated management plane is configured in parallel with the test-bed. It is realised as `192.168.20.0/24`, anchored by a TP-Link device acting as switch, router, wireless AP and DHCP server. All the nodes are reachable via SSH from a bastion host, which has a public IP address. Authentication uses an SSH private key (no password-based authentication is used on the bastion host). After connecting to the bastion, it is possible to reach the internal nodes over the control network without exposing the laboratory machines to the Internet. The DHCP server of the TP-Link device pins `192.168.20.x` subnet to device MAC addresses dedicated to control functions to ensure stable addressing across reboots. This approach simplifies orchestration and makes automation through scripts easier. The overall layout is illustrated in figure 4.4 and the specific devices connected to the control plane are listed in table 4.2.

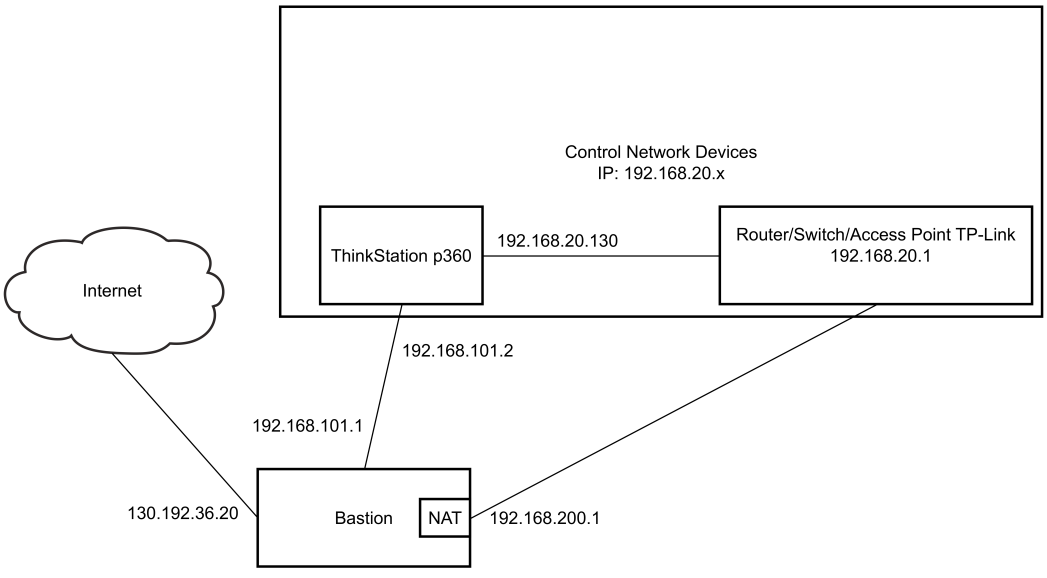


Figure 4.4: Control plane

Table 4.2: Devices connected to the control network (`192.168.20.0/24`)

Device	IP (control)	Control interface
Router/Switch/AP TP-Link	192.168.20.1	Ethernet & Wi-Fi
Lenovo ThinkStation P360 Tower	192.168.20.130	Wi-Fi
Raspberry Pi 2 Model B	192.168.20.131	Ethernet
Raspberry Pi 5 Model B	192.168.20.132	Wi-Fi
Raspberry Pi 3 Model B	192.168.20.133	Wi-Fi
Raspberry Pi 3 Model B	192.168.20.134	Wi-Fi
Raspberry Pi 5 Model B	192.168.20.135	Wi-Fi
Raspberry Pi 5 Model B	192.168.20.136	Wi-Fi
Lenovo ThinkStation P5	192.168.20.137	Ethernet

Remote-access workflow

The bastion node exposes three wired interfaces:

- One interface with a public IP (i.e., connected directly to the Internet).
- One interface toward the TP-Link router. The bastion provides access to Internet to 192.168.20.0/24 via `iptables` `MASQUERADE` (NAT) on the `POSTROUTING` chain of the `nat` table:

```
Chain POSTROUTING (policy ACCEPT 163 packets, 13035 bytes)
pkts bytes target     prot opt in  out        source      destination
31017 11M  MASQUERADE all  --  *   enp0s31f6  0.0.0.0/0  0.0.0.0/0
```

- One point-to-point link with the Lenovo ThinkStation P360 over 192.168.101.0/30, where the bastion is 192.168.101.1 and the P360 is 192.168.101.2.

To access a device on the control network:

1. From a remote machine, connect to the bastion: `ssh guest@130.192.36.20`
2. From the bastion, connect to the P360: `ssh 192.168.101.2`
3. From the P360, hop to the target: e.g., `ssh 192.168.20.132`

4.4 Limitations and reproducibility considerations

The server-zone subnet 10.0.3.0/24 allows direct observation of delivery performance, latency and packet loss that affects applications. For this reason, all experiments employ a capture point on the central router, specifically on the interface assigned the address 10.0.3.254. This location was chosen because it aggregates both ingress traffic destined for the servers and egress responses leaving the zone, thereby yielding a unified perspective on end-to-end service delivery.

The process of creating namespaces and assigning interfaces, configuring routes and installing iptables policies runs through various scripts that establish a clean, identical starting point for each run.

Some limitations remain: the test-bed combines single-board computers and workstation-grade hardware, so CPU and NIC performance differ significantly among the devices; this is a key aspect that must be considered during the experiments.

Chapter 5

Experiments and demos

Chapter 4 describes the setup of the experimental test-bed; the following one concentrates on the actual measurement process. The experiments start with iPerf3 baseline tests, which only verify the proper functioning of the infrastructure under TCP and UDP traffic conditions. The first set of tests confirms network size and reachability, but the iPerf3 bursty flows do not match real application traffic behaviour.

To obtain more representative traffic, the baseline is then repeated with custom Python scripts. The systems perform basic client-server operations through pre-programmed fixed-length request transmissions, which produce standard communication patterns. The method produces network traffic patterns that duplicate real-world systems.

Once the infrastructure is validated under realistic traffic, the firewalls are activated and rules are deployed. Each experiment was conducted twice: once with a static distribution of rules and again after applying the REDIAL algorithm, already introduced in chapter 2. The comparison method enables to determine the performance benefits.

In the baseline experiments, there is a well-defined sequence of phases: normal traffic begins at $t = 0$ s, an ICMP flood is introduced between $t = 15$–45 s and all flows terminate at $t = 60$ s. When the firewalls are activated, the same three phases are stretched by a factor of ten: normal traffic from $t = 0$–150 s, ICMP flood from $t = 150$–450 s and termination at $t = 600$ s. The three observation windows – before, during and after the attack – allow system behaviour monitoring under normal operating conditions, high stress and recovery states.

5.1 Preliminary measurements with iPerf3

Before fulfilling the thesis demo objective, the network behaviour is monitored with iPerf3 while the firewalls act as pass-through devices. (i.e., all the chains of the

filter table of iptables have as default policy ACCEPT and there are no rules). The baseline represents the maximum performance potential the hardware and topology can deliver and it serves as a reference to measure firewall effects.

This section aims to show, through four experiments, how ICMP packets affect legitimate TCP and UDP flows. It serves only as a **preliminary introduction** and these experiments are explored in greater depth in later sections of the chapter using more precise methodologies because iPerf3 has automatic mechanisms that do not allow the manipulation of some packet parameters, such as packet length and sending frequency.

Experimental setup

Figure 5.1 illustrates the topology used for the baseline measurements: the rounded nodes are the endpoints (senders and receivers). Coloured links indicate three traffic types: red represents adversarial traffic, green represents legitimate traffic and orange represents mixed flows. The arrow directions represent the flows from clients to servers. The black links exist in the topology but remain inactive during these experiments. The firewalls indicated are pass-through devices in these baseline measurements.

The risk that the NICs drop packets under high PPS conditions (and particularly during small-ICMP scenarios) is minimized by writing capture files to a `tmpfs` [24] filesystem. Listing 5.1 shows the preparation of a 10 GiB RAM disk for high-rate captures.

```
sudo mkdir /mnt/ramdisk
sudo mount -t tmpfs -o size=10G tmpfs /mnt/ramdisk
```

Listing 5.1: RAM disk for high-rate captures (tmpfs 10 GiB).

Traffic scenarios and timing

The legitimate traffic is produced by running iPerf3 between 10.0.2.1 and 10.0.3.21 using either UDP or TCP. The ICMP flood attack originates from the attacker node 9.0.0.1 that sends its traffic to 10.0.3.21. In this experimental campaign, ICMP Echo traffic – transmitted by the attacker at the maximum achievable rate in every trial – used two payload sizes:

- **0 B ICMP payload:** the IP total length is 28 B. Ethernet therefore pads the payload to 46 B, then the frame size is 64 B (14 B L2 header + 46 B payload + 4 B FCS). Wireshark typically reports about 42 B (14 B L2 + 28 B IP) because padding and the FCS are not provided to the capture stack. In addition, the physical overhead must be added (8 B preamble/SFD + 12 B IFG), so the wire-time per frame is 84 B.

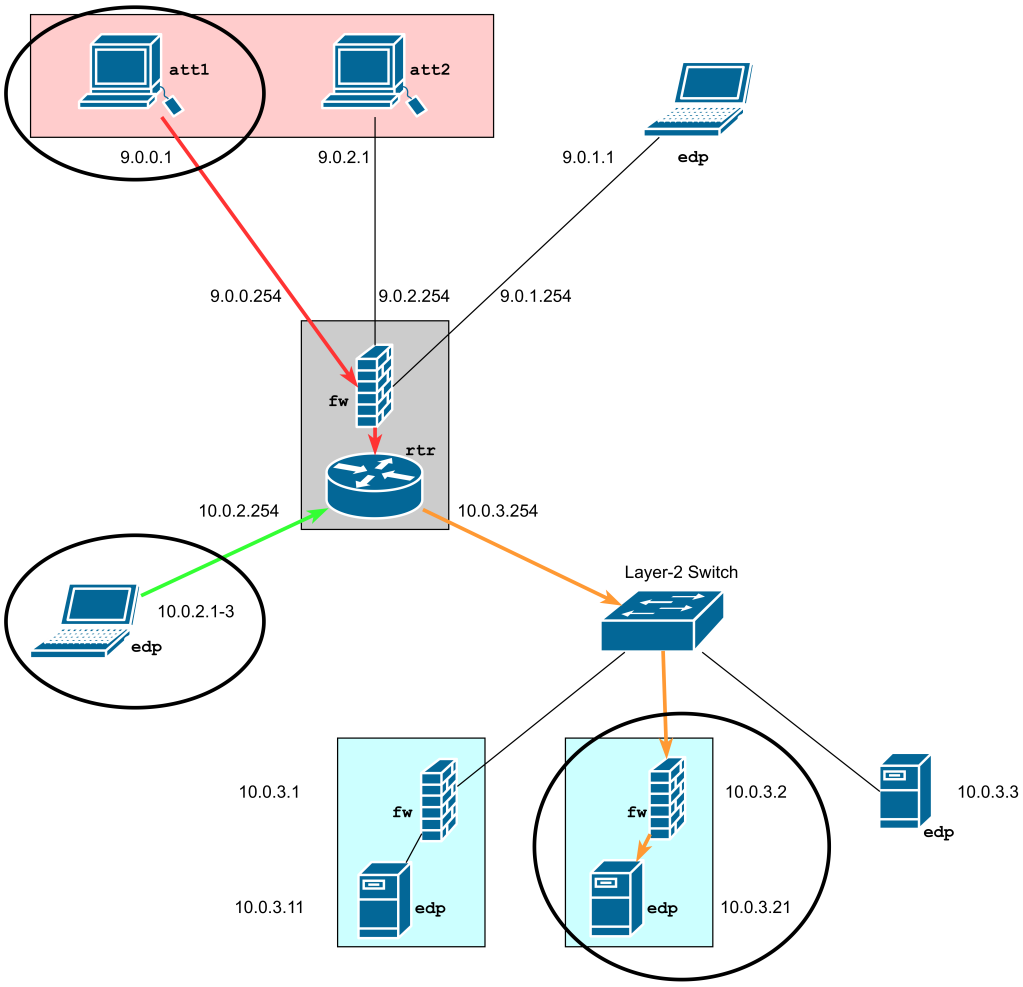


Figure 5.1: Topology for the experiments related to the baseline measurements with iPerf3

- **1400 B ICMP payload:** the IP total length is 1428 B. The corresponding Ethernet frame is 1446 B (14 B L2 + 1428 B + 4 B FCS). The wire-time length is 1466 B (Ethernet frame 1446 B + preamble/SFD 8 B + IFG 12 B)

The "LEN" keyword indicates the packet size (`ip.len` field in Wireshark). Only four measurements were taken, as the goal here was to observe how ICMP payload size roughly affects legitimate traffic flows. Specifically, it is analysed a 2×2 design pairing the transport protocol (TCP and UDP) with the attacker's ICMP payload size (small and large). Table 5.1 summarises the four no-firewall baseline configurations (protocol × ICMP size). All graphs presented are obtained using the Pandas filter
`df[(df['ip.src'].isin(['10.0.2.1', '10.0.3.21'])) & (df['ip.dst'].isin(['10.0.2.1', '10.0.3.21']))]`, so they include only the legitimate traffic.

Table 5.1: No-firewall baseline configurations: legitimate L4 protocol × attacker ICMP payload size

	UDP (iPerf3)	TCP (iPerf3)
ICMP 0 B	ICMP 0 B + UDP	ICMP 0 B + TCP
ICMP 1400 B	ICMP 1400 B + UDP	ICMP 1400 B + TCP

Each experiment lasts 60 seconds and has three phases: **Before** (15 seconds) with only iPerf3 traffic, **During** (30 seconds) with the ICMP flood added and **After** (15 seconds) when the flood stops and only iPerf3 traffic continues.

Results by configuration

0 B ICMP payload with UDP (iPerf3)

The analysis, whose results are summarized in table 5.2, indicates that the packets per second (PPS) remain stable across all three observation phases. Similarly, the throughput metric shows no significant variation during the entire measurement. This overall stability is considered an empirical observation that points to a consistent pattern in the generation and transmission of UDP traffic, largely unaffected by the presence of concurrent ICMP activity.

The flood traffic and the legitimate UDP flow enter the router through different ingress interfaces. This configuration may, in principle, trigger the kernel's load-balancing mechanisms: Receive Packet Steering (RPS), Receive Flow Steering (RFS), Transmit Packet Steering (XPS), multiqueue drivers and queuing disciplines (qdisc) can affect how packets are distributed across cores and thus alter processing latency. In this setup, however, such parameters were left at their default values and not explicitly audited, so the measurements reflect the behaviour of an unmodified kernel configuration.

Table 5.2 reports the mean and variance of PPS, throughput and packet length for the three phases. The complete iPerf3 output is provided in table A.1 and figure 5.2 illustrates the evolution of these metrics over time.

Table 5.2: Mean and variance of UDP traffic metrics (with 0 B ICMP) across the three phases (before, during and after)

Metric	Before (mean/var)	During (mean/var)	After (mean/var)
PPS [kPPS]	77.68/15.96	77.69/16.03	77.68/13.76
Thput [Mbit/s]	917.18/2245.35	917.41/2235.09	917.26 /1935.40
LEN [Bytes]	1475.69/413.29	1476.00/0.00	1475.81/255.30

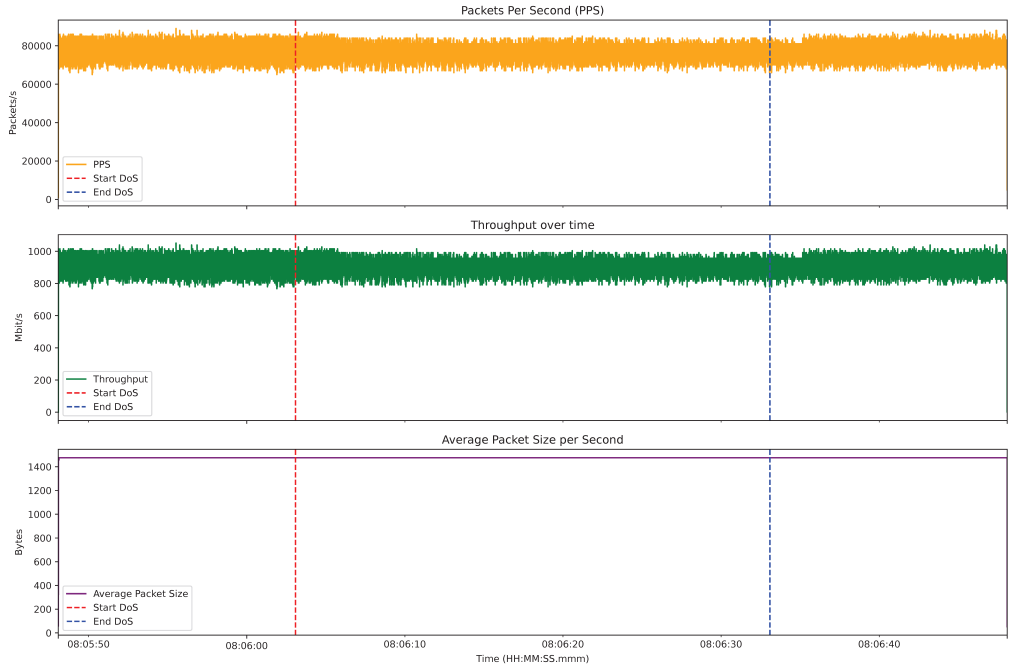


Figure 5.2: Traffic metrics (PPS, throughput and LEN) over time in the no-firewall baseline with UDP and 0 B ICMP payload, across the three experimental phases (before, during and after)

0 B ICMP payload with TCP (iPerf3)

In this case, a complete connection outage is initially observed (there is a simultaneous collapse of all traffic metrics), but after a short period, the communication is restored; however, performance remains severely degraded due to a high number of retransmissions. Figure 5.3 shows PPS, throughput and LEN across the three phases under the 0 B ICMP flood, while table 5.3 reports mean/variance for TCP metrics under the 0 B ICMP flood across phases. The complete iPerf3 results are provided in table A.2.

The PPS panel of figure 5.3 shows that the baseline rate stabilizes near 100 kPPS but collapses to close to zero at attack onset; the throughput curve also falls from >950 Mbit/s to below <100 Mbit/s and displays high variance in the third phase. During the attack phase, the LEN metric indicates that the legitimate packets sent by iPerf3 are smaller on average, with the mean size falling by more than 50% and the variance exploding, consistently with frequent retransmissions. Overall, the data indicate that small ICMP packets cause degradation to TCP communication by triggering excessive retransmissions and reducing the average packet size.

The ICMP flood in this experiment relies on very small ICMP packets; so the victim must handle an extremely high packet rate and the resulting per-packet costs overhead dominates system performance. Each small-sized packet requires header

parsing at L3/L4, hardware interrupt handling and context switches, processing through of the OS networking stack and CPU work for protocol processing. As these operations accumulate, interrupt handling and software resources processing become saturated, not the raw link capacity. The system reaches the limits of its packet processing capabilities, leading to the TCP stack into timeouts and retransmission. In the third phase of the experiment (the one where the flood traffic ceases), the system rapidly returns to normal operation, with throughput, packet rate and average packet size recovering to baseline values.

Table 5.3: Mean and variance of TCP metrics under 0 B ICMP flood

Metric	Before (mean/var)	During (mean/var)	After (mean/var)
PPS [kPPS]	98.80/13.38	9.37/180.34	98.92/12.92
Thput [Mbit/s]	977.38/1134.20	93.99/19361.64	977.07/1140.92
LEN [Bytes]	1236.60/711.53	574.23/384046.72	1234.94/513.73

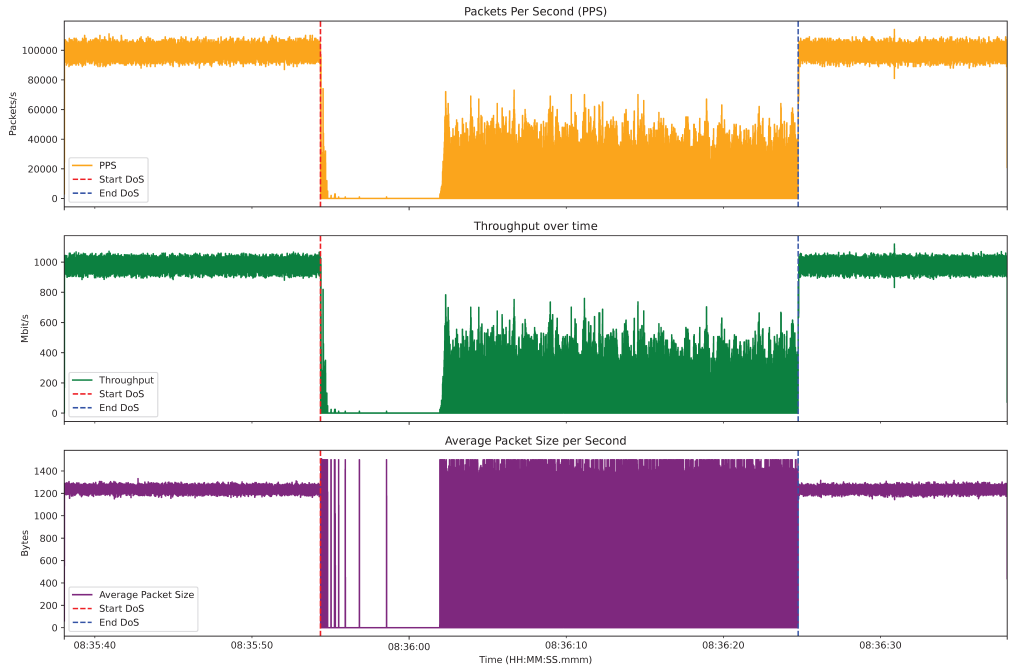


Figure 5.3: Traffic metrics (PPS, throughput and LEN) over time in the no-firewall baseline with TCP and 0 B ICMP payload, across the three experimental phases (before, during and after)

1400 B ICMP payload with UDP (iPerf3)

In this scenario – UDP traffic generated by iPerf3 is influenced by a stream of 1400 B ICMP payload packets – the connection is not interrupted but experiences partial degradation during the attack period. Figure 5.4 depicts the evolution of PPS, throughput and LEN with a 1400 B ICMP flood; while table 5.4 shows mean/variance for UDP metrics. The full iPerf3 output is shown in table A.3.

During the attack, a reduction in PPS and throughput metrics is observed and the average packet size remains constant and its variance drops to zero, resulting in an overall loss of approximately 48%, as can be seen in table A.3. The graphs indicate that both PPS and throughput stabilize at lower values for the entire attack interval and the average length curve remains perfectly flat; this indicates that UDP traffic continues to be transmitted uniformly (even if at a lower rate).

Transmission is rate-limited rather than interrupted. Packet length remains constant, consistent with an unchanged iPerf3 sending pattern under stress. The flow continues, but PPS and throughput fall substantially. The ICMP flood creates contention; after it stops PPS and throughput immediately revert to baseline, as can be observed in figure 5.4.

Table 5.4: Mean and variance of UDP traffic metrics (with 1400 B ICMP) across the three phases (before, during and after)

Metric	Before (mean/var)	During (mean/var)	After (mean/var)
PPS [kPPS]	77.68/18.00	41.95/13.72	77.93/17.12
Thput [Mbit/s]	917.24/2528.17	495.38/1913.24	920.22/2404.85
LEN [Bytes]	1475.80/252.60	1476.00/0.00	1475.81/264.67

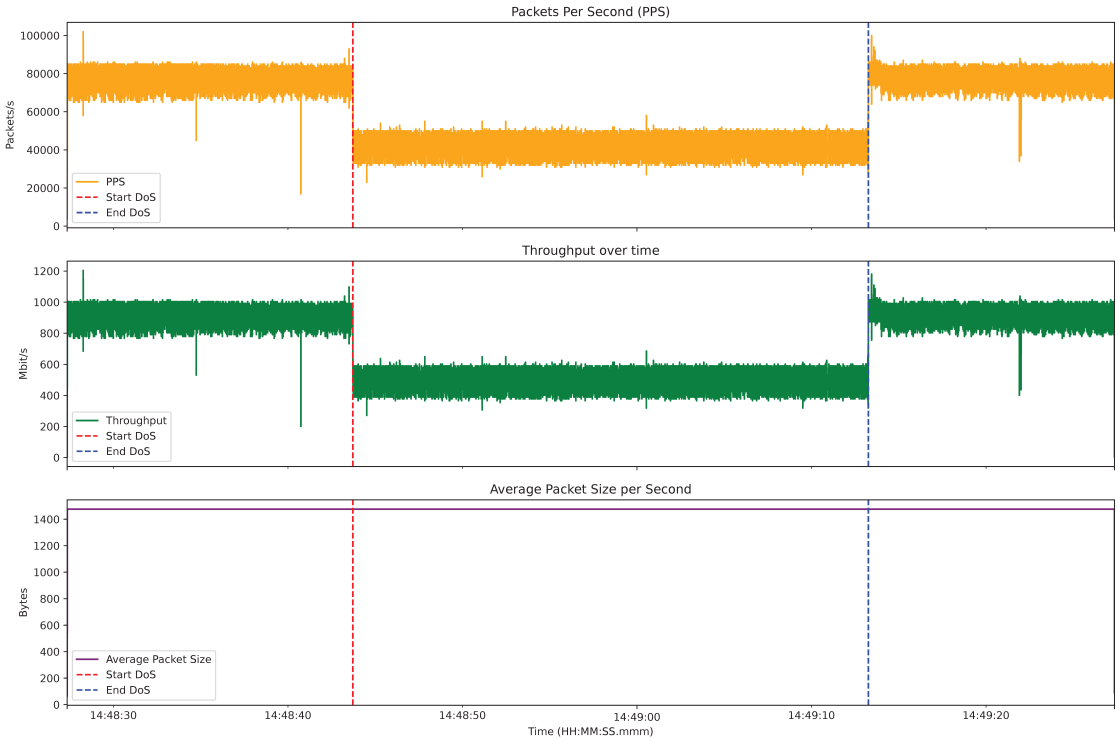


Figure 5.4: Traffic metrics (PPS, throughput and LEN) over time in the no-firewall baseline with UDP and 1400 B ICMP payload, across the three experimental phases (before, during and after)

1400 B ICMP payload with TCP (iPerf3)

In this scenario, the TCP traffic generated by iPerf3 is present with a stream of large packets that have 1400 B ICMP payload (1466 B on-wire). The attack does not cause an immediate connection outage; but it results in an anomalous TCP behaviour: the traffic shows instability and performance degradation. Figure 5.5 reports PPS, throughput and LEN during the 1400 B ICMP flood, while table 5.5 summarises TCP metrics across the three phases.

The attack phase shows an increase in packet rate (PPS) with a marked drop in throughput and a substantial reduction in average packet size. The variance is high in all the metrics (particularly in throughput and packet length). This behaviour indicates a highly unstable TCP traffic that is probably caused by the retransmissions and network congestion triggered by the ICMP attack, similarly to the previously discussed case of the 0 B ICMP payload attack with TCP legitimate traffic.

Table A.4 lists the detailed iPerf3 output for this experiment.

Table 5.5: Mean and variance of TCP traffic metrics (with 1400 B ICMP) across the three phases (before, during and after)

Metric	Before (mean/var)	During (mean/var)	After (mean/var)
PPS [kPPS]	98.84/13.10	123.44/361.86	104.00/200.47
Thput [Mbit/s]	977.29/1080.72	779.11/29839.89	971.22/4703.70
LEN [Bytes]	1235.97/888.77	782.45/11472.97	1180.89/17132.54

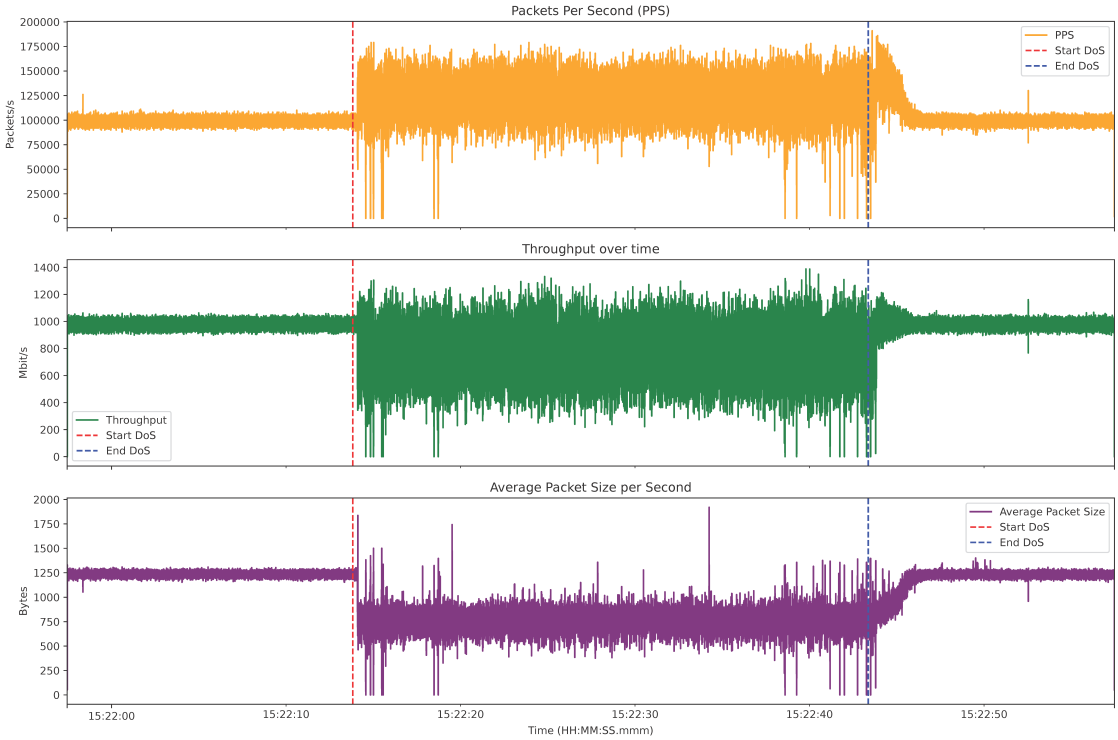


Figure 5.5: Traffic metrics (PPS, throughput and LEN) over time in the no-firewall baseline with TCP and 1400 B ICMP payload, across the three experimental phases (before, during and after)

Results summary

The attack impact depended on both the L4 protocol and the size of the attacker packets. The router's load-balancing mechanism mitigated the small attacker packets (0 B ICMP payload) when combined with the UDP traffic, but it degraded TCP performance. The large attacker packets (1400 B ICMP payload) caused UDP throughput reduction but not a connection disruption; however with TCP experienced significant instability.

5.2 Baseline measurements with custom scripts

The first set of experiments takes place in a clean condition because no filtering rules are applied: all intermediate devices forward packets transparently. The test-bed measures two aspects: it defines the thresholds at which the network becomes unstable and it demonstrates how unprotected systems become vulnerable to ICMP flooding attacks, which VEREFOO and REDIAL will later protect against.

The experiments reuse the test-bed described in chapter 4. Instead of iPerf3, Python scripts generate application traffic: this change was made because iPerf3 includes automatic mechanisms that adjust its behaviour (e.g., packet length or sending frequency—based on network conditions), resulting in a bursty traffic pattern. Two clients establish legitimate flows with two Raspberry Pi servers (10.0.3.11 and 10.0.3.21) through TCP and UDP protocols at specified sizes and intervals. Adversarial traffic is sent by two attacker namespaces: 9.0.0.1 floods 10.0.3.11, while 9.0.2.1 floods 10.0.3.21.

The legitimate traffic is represented by the two connections established via a Python script: one between 9.0.1.1 and 10.0.3.21 and the other between 9.0.1.1 and 10.0.3.11. Each segment (TCP) or datagram (UDP) has a fixed size payload of 500 B. The traffic is sent at a constant rate of 20 requests per second (or less if the network is congested. For each request, the client script waits either for a response or until a timeout expires before sending the next request).

As in the previous experiments of the section 5.1, all traffic is captured on the central router on the interface with IP 10.0.3.254 with hardware timestamping and buffered on a RAM-disk to minimize overhead. Also, regarding the timing of the experiment, the phases are the same as the baseline experiments with iPerf3 illustrated in section 5.1.

Figure 5.6 illustrates the topology used in these measurements. As in the section 5.1, the rounded nodes are the endpoints (senders and receivers). Coloured links indicate three traffic types: red represents adversarial traffic, green represents legitimate traffic and orange represents mixed flows. The arrow directions represent the flows from clients to servers. The black links exist in the topology but remain inactive during these experiments. The firewalls indicated are pass-through devices in these baseline measurements.

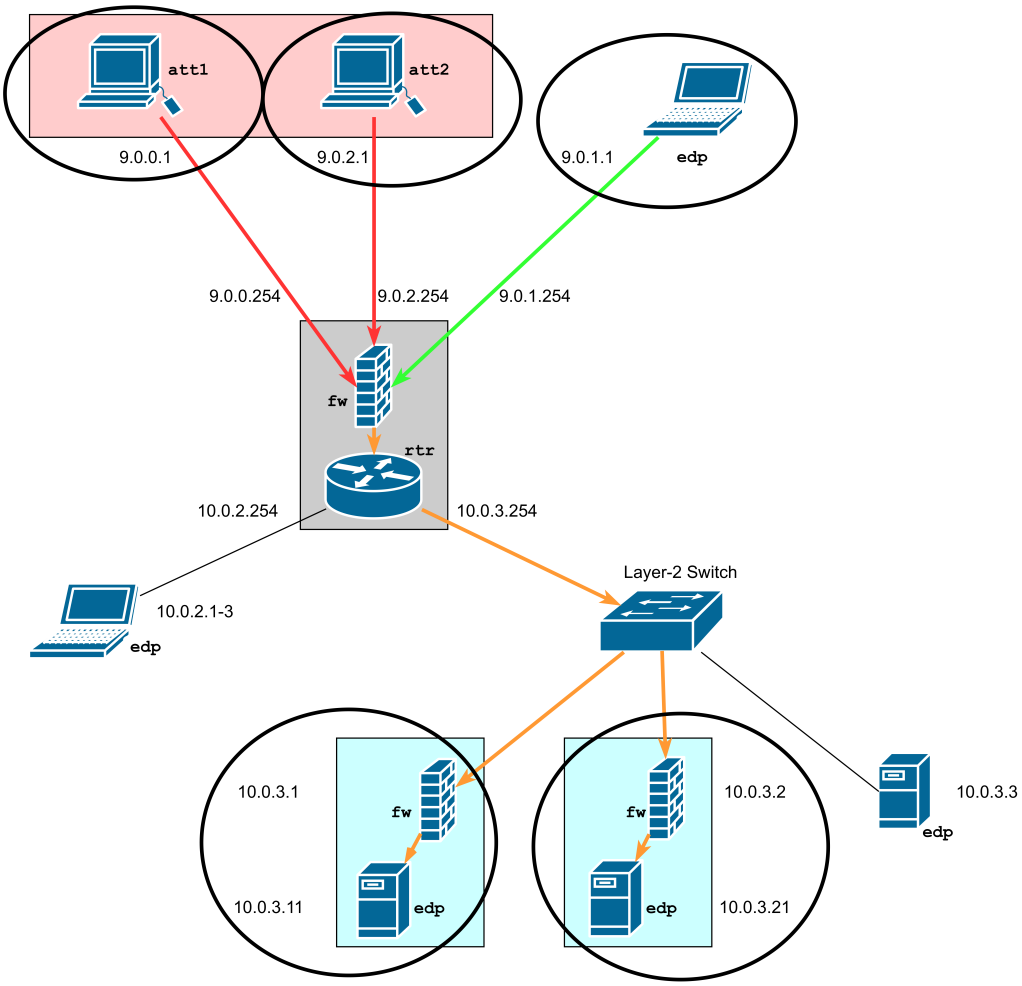


Figure 5.6: Topology for baseline measurements with custom scripts and for the final demo

Results

To analyse the effect of ICMP flooding on legitimate traffic, the results are presented in terms of two key values:

- The **degradation threshold** represents the largest packet size which an attacker can send to make the victim traffic experience major performance degradation. For UDP, this is the packet size at which loss falls below 10%; for TCP, it is the size at which retransmissions remain fewer than 200.

- The **safe threshold** is the ICMP size above which no significant slowdowns are noticeable. For UDP, this is the packet size at which loss falls below 2%; for TCP, it is the size at which retransmissions remain fewer than 50.

By expressing the results in this way, it becomes possible to compare UDP and TCP behaviour directly and to identify the attacker traffic characteristics that most endanger system stability. The thresholds are shown in the table 5.6.

Table 5.6: Observed thresholds summary

Protocol	Degradation threshold	Safe threshold
UDP	$<$ 150 B	$>$ 270 B
TCP	$<$ 130 B	$>$ 200 B

UDP

As shown in Figure 5.7, UDP traffic is highly vulnerable to ICMP flooding when the attack payload size is small (e.g., 0–100 B), with loss percentages exceeding 40%. But starting from approximately 150 Bytes, a marked reduction in loss can be observed (it drops below 10%). Above 270 Bytes, the disruption becomes negligible or completely absent.

These two thresholds can help us to delineate two zones: a critical zone below 150 Bytes where UDP flows suffer heavy degradation and a safe zone beyond 270 Bytes where the legitimate traffic is unaffected. These results can serve as a reference point for further analysis of firewall configurations. Table 5.7 lists UDP requests/responses by phase and host for each payload level.

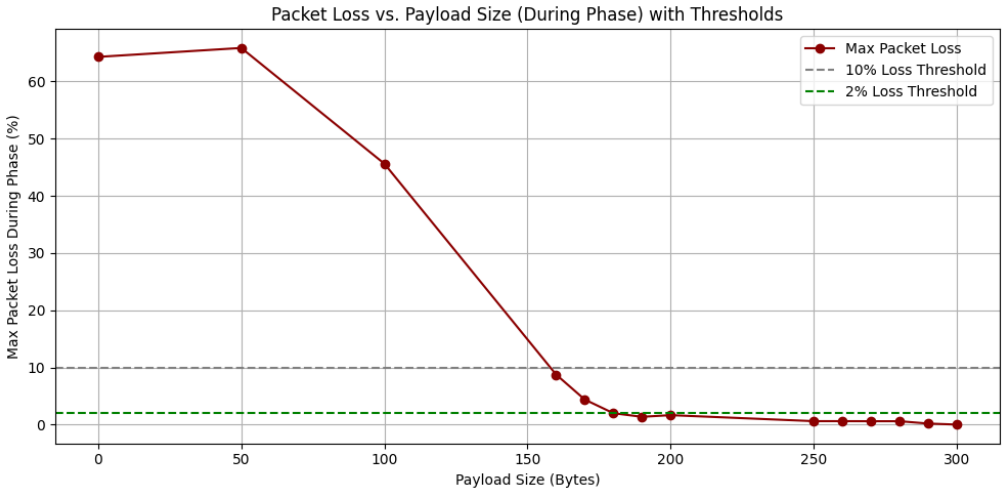


Figure 5.7: UDP Packet loss vs Payload size during ICMP flood attack

Table 5.7: UDP requests/responses per phase and host

Level	Payload (B)	Host	Before		During		After	
			Requests	Responses	Requests	Responses	Requests	Responses
1	50	10.0.3.11	306	306	50	23	269	268
1	50	10.0.3.21	306	306	41	14	279	278
2	100	10.0.3.11	300	300	57	31	286	285
2	100	10.0.3.21	300	300	72	46	269	268
4	160	10.0.3.11	301	301	206	188	280	279
4	160	10.0.3.21	301	301	217	199	269	268
5	170	10.0.3.11	300	300	297	284	280	279
5	170	10.0.3.21	300	300	335	324	278	277
6	180	10.0.3.11	300	300	419	412	269	268
6	180	10.0.3.21	300	300	403	395	288	288
7	190	10.0.3.11	301	301	459	454	288	288
7	190	10.0.3.21	301	301	440	434	269	268
8	200	10.0.3.11	301	301	425	418	269	268
8	200	10.0.3.21	301	301	427	420	269	268
9	250	10.0.3.11	300	300	502	499	288	288
9	250	10.0.3.21	300	300	502	499	269	268
10	260	10.0.3.11	301	301	504	501	288	288
10	260	10.0.3.21	301	301	540	539	288	288
11	270	10.0.3.11	300	300	557	557	288	288
11	270	10.0.3.21	300	300	505	502	289	289
12	280	10.0.3.11	301	301	505	502	288	288
12	280	10.0.3.21	301	301	525	523	269	268
13	290	10.0.3.11	300	300	541	540	288	288
13	290	10.0.3.21	300	300	562	562	268	267
14	300	10.0.3.11	300	300	560	560	269	268
14	300	10.0.3.21	300	300	562	562	270	269

Figure 5.8 shows the number of UDP requests received during the attack as a function of the ICMP packet size. The total number of received UDP packets—aggregated across hosts 10.0.3.11 and 10.0.3.21—increases sharply with larger ICMP payloads.

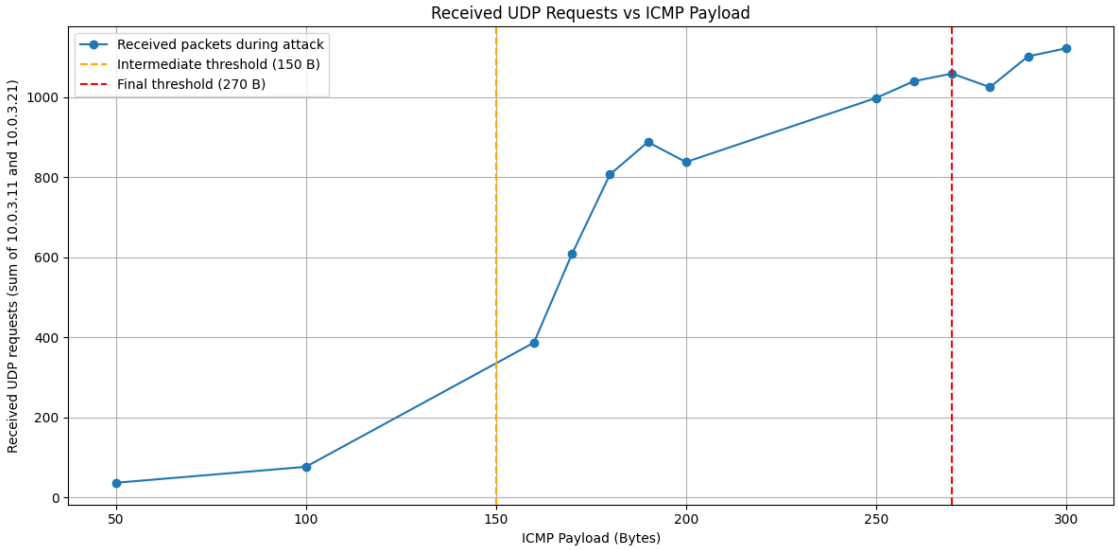


Figure 5.8: Received UDP Requests vs ICMP Payload size

TCP

The TCP case shows two main metrics, which include the **average Round-Trip Time (RTT)** and the **number of TCP retransmissions**. These data are extracted using `tshark` filters `tcp.analysis.ack_rtt` and `tcp.analysis.retransmission`. Figure 5.9 illustrates that the average RTT during the attack decreases as the ICMP packet size increases. For packets smaller than approximately 130 Bytes the RTT values rise sharply: this is an indication of a severe degradation of network responsiveness. Between 130 and 200 Bytes, the RTT values during the attack gradually decrease. The RTT remains stable at below 10 ms in both phases when the ICMP packet size exceeds 200 Bytes.

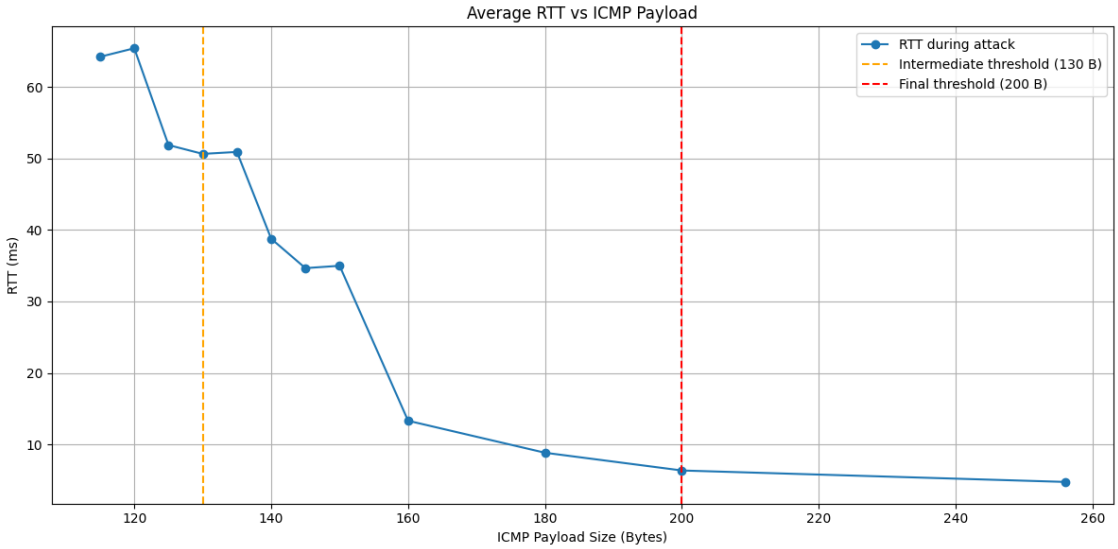


Figure 5.9: Average RTT vs ICMP Payload size during the second phase

Figure 5.10 shows the number of TCP retransmissions during the attack. Retransmissions induce the most severe disruption in the range of 115–130 Bytes. When the packet size exceeds 200 Bytes, the network shows a clear recovery in terms of retransmission behaviour.

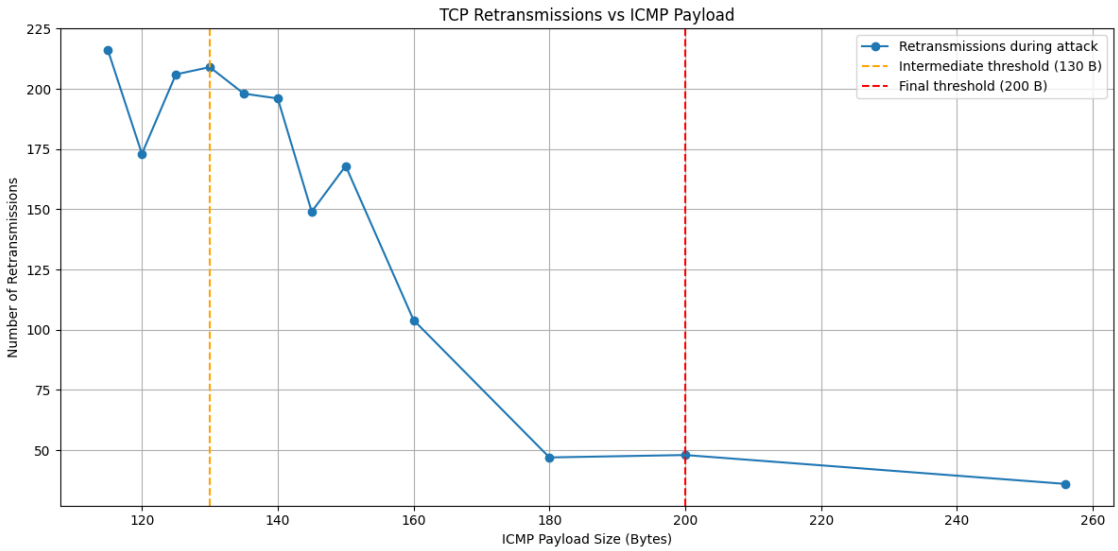


Figure 5.10: TCP Retransmissions vs ICMP Payload size during the second phase

Table 5.8 shows the TCP statistics with the number of retransmissions, DupACK (`tcp.analysis.duplicate_ack` `tshark` filter), loss (`tcp.analysis.lost_segment` `tshark` filter) and other data obtained with Python and `tshark` used together to

analyse the PCAP files. The levels from 1 to 4 are not visualized because of the significant packet loss, which makes measurements in those cases unreliable.

Table 5.8: TCP Packet Statistics

Level	TCP	Retrans	DupACK	Loss %	Avg RTT (ms)	ICMP Size
1	–	–	–	–	–	50 B
2	–	–	–	–	–	100 B
3	–	–	–	–	–	105 B
4	–	–	–	–	–	110 B
5	4898	219	34	4.47	16.68	115 B
6	4866	176	21	3.62	16.35	120 B
7	5063	212	30	4.19	15.76	125 B
8	5165	212	28	4.1	15.09	130 B
9	5136	199	38	3.87	14.69	135 B
10	5433	198	39	3.64	12.78	140 B
11	5425	158	19	2.91	12.44	145 B
12	5506	169	27	3.07	11.96	150 B
13	4672	120	24	2.57	13.33	160 B
14	4763	48	15	1.01	8.86	180 B
15	4711	51	12	1.08	6.38	200 B
16	4693	37	14	0.79	4.78	256 B

Table 5.9 shows instead the requests sent and received by the client. The data from this table is obtained through the logs of the client script. It is noticeable that the loss is 0 because TCP is a reliable protocol, so to measure the actual data loss, the study relies on the PCAP data that shows the actual retransmissions. These results are visualised in figure 5.11.

Table 5.9: TCP requests/responses per phase and host

Level	Payload (B)	Host	Before		During		After	
			Requests	Responses	Requests	Responses	Requests	Responses
1	50	10.0.3.11	311	311	15	15	5	5
1	50	10.0.3.21	311	311	61	61	241	241
2	100	10.0.3.11	311	311	147	147	286	286
2	100	10.0.3.21	311	311	15	15	286	286
5	110	10.0.3.11	311	311	166	166	286	286
5	110	10.0.3.21	311	311	208	208	276	276
6	120	10.0.3.11	311	311	235	235	285	285
6	120	10.0.3.21	311	311	143	143	289	289
7	125	10.0.3.11	310	310	230	230	270	270
7	125	10.0.3.21	310	310	227	227	279	279
8	130	10.0.3.11	311	311	233	233	283	283
8	130	10.0.3.21	311	311	235	235	286	286
9	135	10.0.3.11	311	311	222	222	289	289
9	135	10.0.3.21	311	311	230	230	289	289
10	140	10.0.3.11	311	311	251	251	289	289
10	140	10.0.3.21	311	311	296	296	285	285
11	145	10.0.3.11	311	311	268	268	290	290
11	145	10.0.3.21	311	311	294	294	286	286
12	150	10.0.3.11	310	310	307	307	285	285
12	150	10.0.3.21	310	310	282	282	289	289
13	160	10.0.3.11	306	306	342	342	294	294
13	160	10.0.3.21	306	306	412	412	290	290
14	180	10.0.3.11	306	306	425	425	293	293
14	180	10.0.3.21	306	306	487	487	290	290
15	200	10.0.3.11	306	306	499	499	291	291
15	200	10.0.3.21	306	306	544	544	287	287
16	256	10.0.3.11	307	307	550	550	293	293
16	256	10.0.3.21	307	307	550	550	290	290

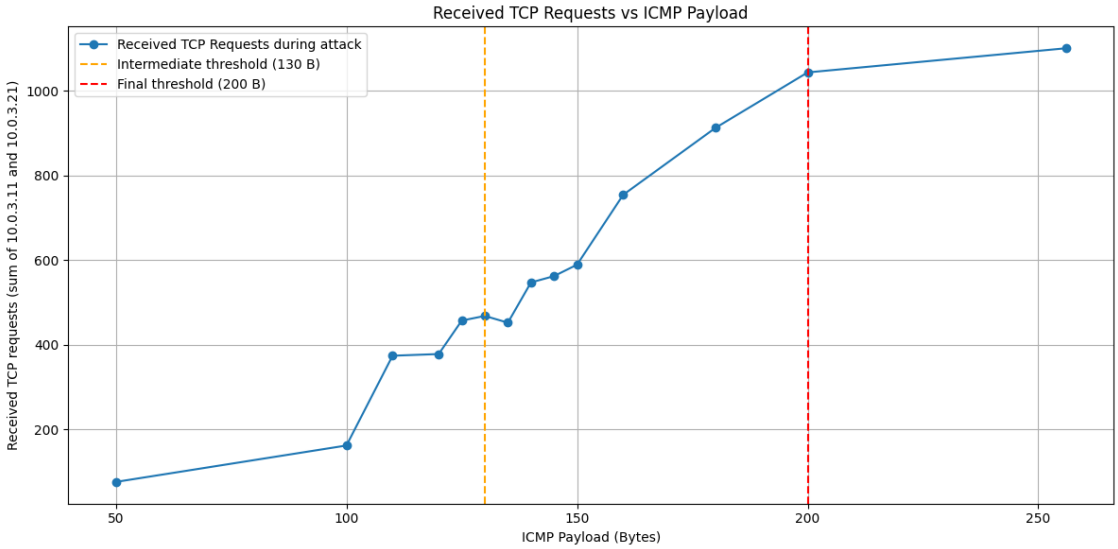


Figure 5.11: Received TCP Requests vs ICMP Payload size

5.3 Final demo

The final demonstration deploys the integrated VEREFOO–REDIAL system. The procedure follows five steps:

1. The Service Graph and the security policies are defined and provided as input to the VEREFOO framework (listing A.2).

2. VEREFOO solves the MaxSMT instance and generates the rule set for the upstream firewall (listing A.3). Since firewalls are stateless, each permitted flow is expressed as a pair of `FORWARD` rules.

3. The resulting rules are loaded, after which the measurement campaigns are executed.

4. REDIAL is then activated, redistributing the rules among the cascaded firewalls and a second campaign is carried out under identical conditions.

5. Finally, the results of the two campaigns are compared.

The methodology with which the experiments are done is the same as the section 5.2, where the firewalls were pass-through elements: same scripts and same setup. There are two main changes: (i) the timing is a 10× longer (2.5 min / 5 min / 2.5 min); (ii) The legitimate traffic has a different bitrate which varies slightly across measurements but is typically around 100–110 Mbps (corresponding to about 10% of a 1 Gbps link) and has a L4 payload (TCP or UDP) of 1400 Bytes.

VEREFOO input and output

The first step consists of creating an XML file that defines the Service Graph and the corresponding security policies, which is provided as input to the VEREFOO framework.

When VEREFOO has solved the MaxSMT problem instance using the AP algorithm, VEREFOO continues its execution by mapping each element with a pair of `FORWARD` rules: one for the request path and one for the response path. This is important because the firewalls considered in this thesis are stateless.

Figure 5.12 compares the real test-bed and its VEREFOO model.

The listing A.2 shows the XML file provided as input to VEREFOO. Reachability properties encode flows that must be allowed (e.g., HTTP, HTTPS, SSH, DB and custom ports); isolation properties encode flows that must be denied (e.g., TCP ports below specific thresholds and ICMP).

The corresponding output is reported in listing A.3, which shows the configuration generated for the upstream firewall 100.0.0.1 with the DENY default action plus a rule set of the smallest possible size.

Each VEREFOO element $\langle \text{src}, \text{dst}, \text{protocol}, \text{src\_port}, \text{dst\_port} \rangle$ is translated into an iptables entry of the form `-A FORWARD -p <proto> -s <src> -d <dst> -sport <src_port> -dport <dst_port> -j ACCEPT`. Since the firewalls are stateless, the corresponding reply rule must also be included: `-A FORWARD -p <proto> -s <dst> -d <src> -sport <dst_port> -dport <src_port> -j ACCEPT`. This translation is shown in listing A.4.

Measurement campaigns with/without REDIAL

After applying the rules to the upstream firewall, a new measurement campaign is carried out. In the table 5.10 are reported the results for the TCP protocol, while for the UDP protocol, the data are presented in the table 5.11. The distinction between sent or received packets is not important because there is a 0% packet loss.

Bash scripts automate the rule deployment over SSH and SCP from the controller. They export the current rules with `iptables-save` and apply updates with `iptables-restore`. For the REDIAL algorithm runs (see section 2.2), a Python tool reads the `iptables-save` file of the upstream firewall and then the program generates one specific rule set for each firewall (upstream and downstream). The automation scripts transfer and restore each iptables configuration file to the respective nodes.

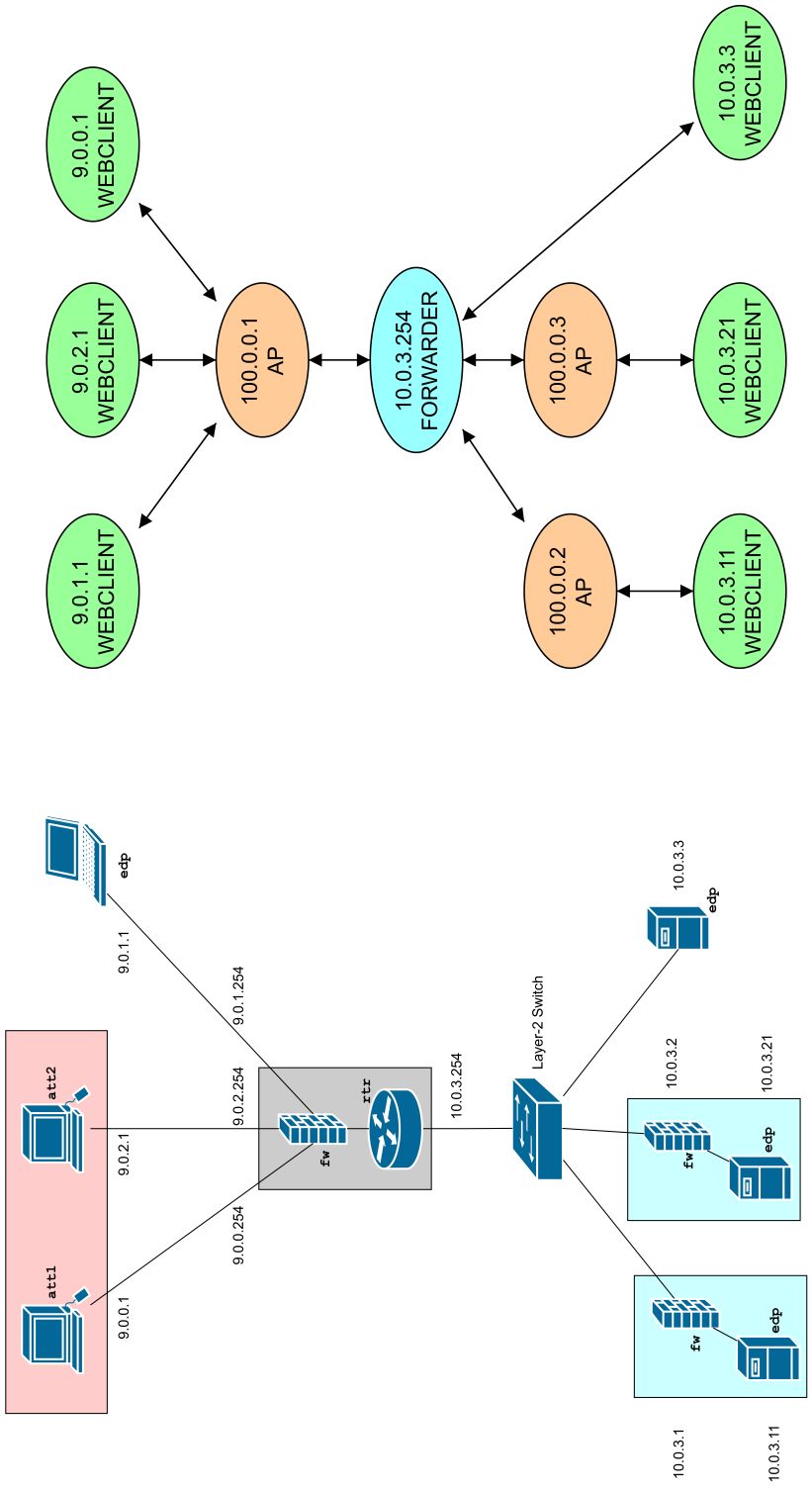
Table 5.10: TCP requests per phase and host with/without REDIAL

Level	ICMP Size (B)	Host	Before		During		After	
			No redial	Redial	No redial	Redial	No redial	Redial
0	0	10.0.3.11	356217	354216	720149	725959	351421	351808
0	0	10.0.3.21	355540	354407	720553	727133	350364	352769
1	50	10.0.3.11	355458	355556	715189	721107	350205	354681
1	50	10.0.3.21	355022	356226	715862	721557	351226	354780
2	100	10.0.3.11	353382	355188	713297	725007	349453	351047
2	100	10.0.3.21	354823	355602	715927	726036	351630	350804
3	105	10.0.3.11	350485	353719	711863	716070	349814	344335
3	105	10.0.3.21	352095	354799	714729	721218	351919	353289
4	110	10.0.3.11	351551	353145	717091	719928	347579	352106
4	110	10.0.3.21	352871	354033	718918	719099	349154	353501
5	115	10.0.3.11	351003	351524	723165	723522	350993	354530
5	115	10.0.3.21	350587	353097	723177	725066	351604	355099
6	120	10.0.3.11	352296	354823	715497	724152	348843	353170
6	120	10.0.3.21	352179	355765	715745	724521	349992	353710
7	125	10.0.3.11	356511	352063	716632	723196	351945	350775
7	125	10.0.3.21	354896	353016	714884	723958	349907	351511
8	130	10.0.3.11	356523	353901	720847	719785	350487	351530
8	130	10.0.3.21	357429	355712	719874	721337	350705	352351
9	135	10.0.3.11	353909	354218	722788	716959	353138	352612
9	135	10.0.3.21	354528	355398	722032	719713	354293	353394
10	140	10.0.3.11	353100	352046	721637	714909	353034	350964
10	140	10.0.3.21	354138	352924	724040	716503	354164	351413
11	145	10.0.3.11	352417	352046	722621	718498	353004	346197
11	145	10.0.3.21	353248	352924	723661	722624	353969	345929
12	150	10.0.3.11	354093	352017	721947	718899	352184	348846
12	150	10.0.3.21	354235	353783	723539	720786	353436	350952
13	160	10.0.3.11	349120	351058	723482	712164	355451	346024
13	160	10.0.3.21	350792	353120	724394	713599	356802	348140
14	180	10.0.3.11	352313	353967	724469	719809	352494	353394
14	180	10.0.3.21	353310	351741	725570	717708	353115	350491
15	200	10.0.3.11	355762	353996	720802	719237	351910	352016
15	200	10.0.3.21	355713	352128	721884	716688	352689	349863
16	256	10.0.3.11	355182	354181	724254	713439	352541	351936
16	256	10.0.3.21	355263	352882	725799	719825	352722	350864

Table 5.11: UDP requests per phase and host with/without REDIAL

Level	ICMP Size (B)	Host	Before		During		After	
			No redial	Redial	No redial	Redial	No redial	Redial
0	0	10.0.3.11	338268	331719	673598	675308	327717	328468
0	0	10.0.3.21	338193	331799	673456	675260	327682	328630
1	50	10.0.3.11	330541	329914	669321	677863	327200	328641
1	50	10.0.3.21	330272	330025	668460	677455	326791	328676
2	100	10.0.3.11	329360	329914	670252	677059	327292	328432
2	100	10.0.3.21	329028	330025	669570	676825	326796	328670
3	150	10.0.3.11	329669	331530	672062	675846	327197	328864
3	150	10.0.3.21	329301	331680	671387	675515	326656	329041
4	160	10.0.3.11	332400	331616	674106	676500	327099	326721
4	160	10.0.3.21	331926	331972	673439	676290	326751	326879
5	170	10.0.3.11	324234	331091	670624	676251	320589	329221
5	170	10.0.3.21	324369	331181	671649	676006	320653	329541
6	180	10.0.3.11	329809	331654	671704	675608	327280	328668
6	180	10.0.3.21	329503	332031	671210	675334	326980	328839
7	190	10.0.3.11	330200	331927	671098	677746	325309	328548
7	190	10.0.3.21	329912	332299	670579	677402	325119	328577
8	200	10.0.3.11	330266	331794	672461	677165	327126	327690
8	200	10.0.3.21	330037	332128	671891	676726	326657	327819
9	250	10.0.3.11	331160	324101	672125	655031	326335	321130
9	250	10.0.3.21	330781	324100	671043	655187	325939	321157
10	260	10.0.3.11	328557	327011	672075	663175	328424	325152
10	260	10.0.3.21	328317	327019	671194	663369	328028	325136
11	270	10.0.3.11	330293	322496	671036	653659	327030	321340
11	270	10.0.3.21	329774	322545	670264	653739	326568	321383
12	280	10.0.3.11	330429	321894	673148	646623	327164	322220
12	280	10.0.3.21	330112	321926	672618	646647	326903	322183
13	290	10.0.3.11	329688	327362	671888	662533	326371	322678
13	290	10.0.3.21	329170	327485	671383	662694	326030	322674
14	300	10.0.3.11	329363	324349	670034	654489	327751	320183
14	300	10.0.3.21	328972	324139	669441	654563	327429	320161

The figures 5.13 and 5.14 represent respectively results for the TCP and UDP protocols during the second phase, where the attack is launched. Results indicate that in both cases the effects of the REDIAL algorithm are consistent under the 8^{th} level ($<$130 B for TCP and $<$200 B for UDP). Therefore, REDIAL is effective for the small packets and this is particularly relevant since the attackers usually rely on small packets to flood their targets, as explained by Zhou et al. [25]. It should be noted that in the demo, a response is always awaited before issuing the next request (or until the request times out), so delays accumulated across the various network nodes affect the results. Future work should address this issue to produce cleaner measurements.



(a) Real implementation

(b) VEREFOO modelling

Figure 5.12: Real deployment and its corresponding VEREFOO representation

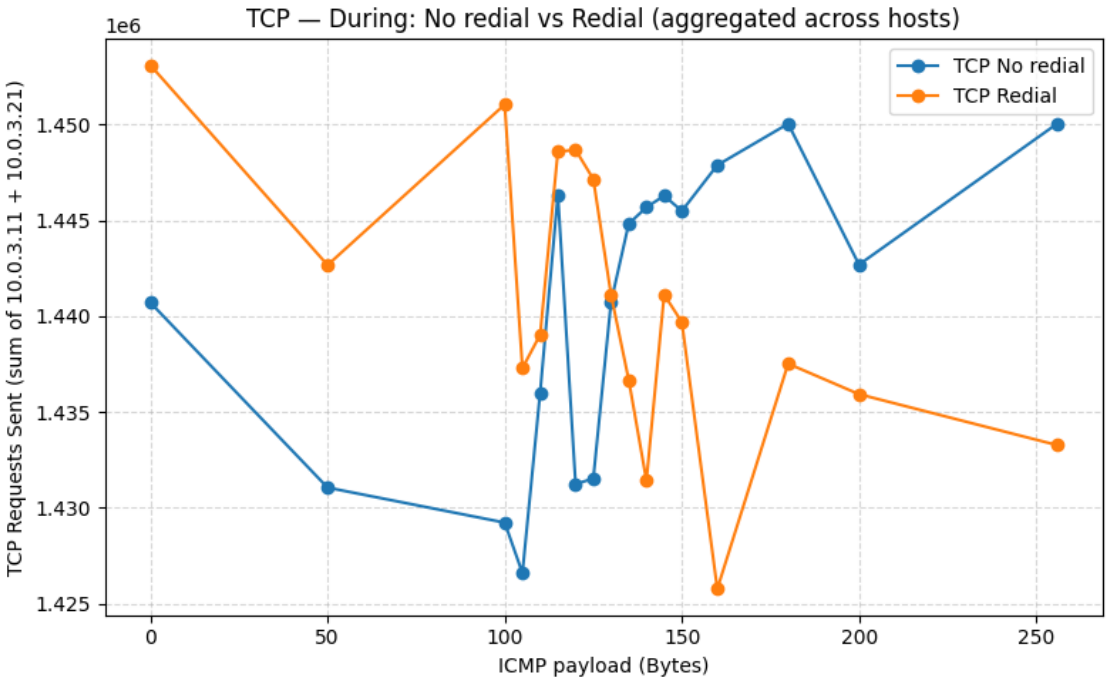


Figure 5.13: Experimental demo campaign: TCP results

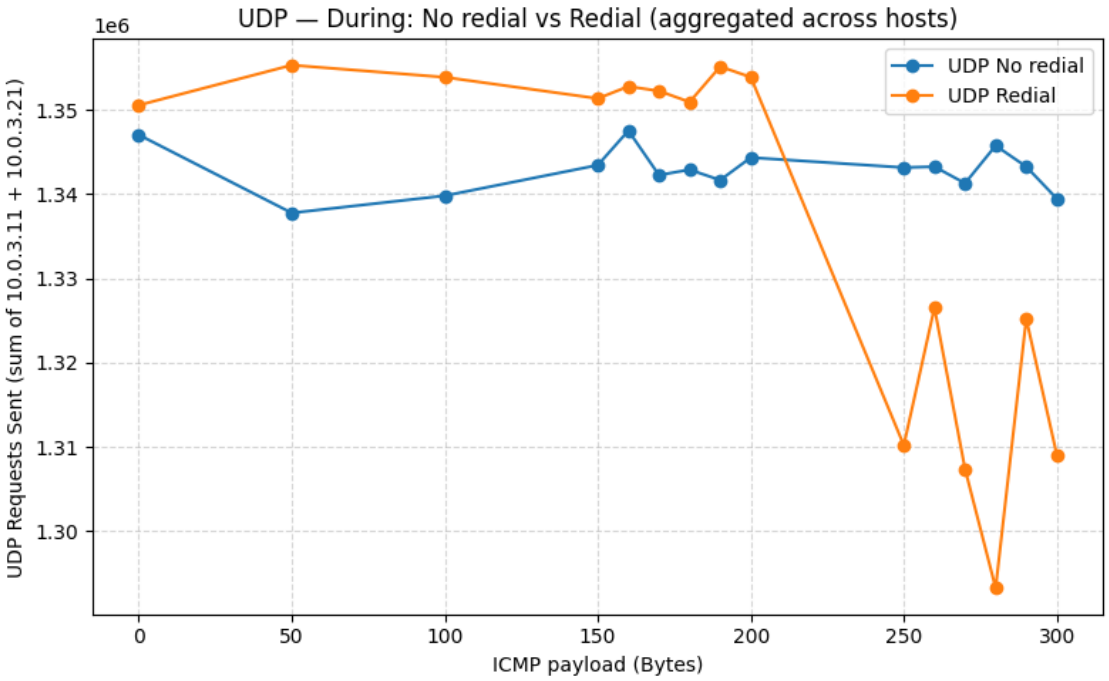


Figure 5.14: Experimental demo campaign: UDP results

Chapter 6

Conclusions

This thesis combined two complementary approaches: VEREFOO, which outputs optimized and formally verified firewall configuration at design-time and REDIAL, which performs rule redistribution at runtime to minimize system overload during unusual traffic pattern such as attacks.

The experiments on the test-bed showed that the integration works in practice. VEREFOO was able to generate valid iptables configurations and REDIAL reduced the load on the upstream firewall at runtime. The results show that this integration improves resilience.

However, there are some limitations: the test-bed hardware is composed of different components that affect system performance and the attack scenarios were limited to ICMP flood attacks. In this work, the system operates with `iptables-legacy`; but most contemporary Linux platforms also implement nftables or eBPF.

Future work could evolve in the following ways: the experiments could be conducted with different attacks and protocols, the integrated VEREFOO-REDIAL approach could be tested on more homogeneous and powerful hardware, modern packet filtering frameworks could be used and it could be explored whether to use REACT-VEREFOO [26], [27] for automated firewall reconfiguration.

In conclusion, this thesis shows that it is possible to join design-time verification solutions with runtime flexibility in a single holistic system. Security technologists are no longer forced to choose between assurance and agility: this integration demonstrates that they can pursue both at once.

Appendix A

Appendix

A.1 Central router setup script

```
#!/bin/bash

user=$(id -u)

if [ "$user" -ne 0 ]; then
    echo "Must be root to run this script."
    exit 1
fi

if [ $# -ne 1 ]; then
    echo "Usage: setup [create|clear]"
    exit 1
fi

if [ "$1" = "create" ]; then
    echo "[INFO] Starting setup: CREATE"

    echo "[1/10] Creating network namespaces 'fw' and 'rtr'..."
    ip netns add fw
    ip netns add rtr

    echo "[2/10] Creating veth pair 'veth-fw' <-> 'veth-rtr'..."
    ip link add veth-fw type veth peer name veth-rtr
    ip link set veth-fw netns fw
    ip link set veth-rtr netns rtr

    echo "[3/10] Moving physical interfaces to namespaces..."
    echo "       Moving enp2s0 and enp4s0 to 'fw'..."
    ip link set enp2s0 netns fw
    ip link set enp4s0 netns fw
    ip link set eno2 netns fw

    echo "       Moving enp1s0f0 and enp1s0f1 to 'rtr'..."
    ip link set enp1s0f0 netns rtr
    ip link set enp1s0f1 netns rtr

    echo "[4/10] Assigning IP addresses to interfaces..."
    echo "       In 'rtr' namespace:"
    ip netns exec rtr ip addr add 10.0.2.254/24 dev enp1s0f0
    ip netns exec rtr ip addr add 10.0.3.254/24 dev enp1s0f1
```

```
    ip netns exec rtr ip addr add 192.168.100.2/30 dev veth-rtr

    echo "       In 'fw' namespace:"
    ip netns exec fw ip addr add 9.0.0.254/24 dev enp2s0
    ip netns exec fw ip addr add 9.0.1.254/24 dev enp4s0
    ip netns exec fw ip addr add 9.0.2.254/24 dev eno2
    ip netns exec fw ip addr add 192.168.100.1/30 dev veth-fw

    echo "[5/10] Bringing up interfaces in 'rtr'..."
    ip netns exec rtr ip link set enp1s0f0 up
    ip netns exec rtr ip link set enp1s0f1 up
    ip netns exec rtr ip link set veth-rtr up
    ip netns exec rtr ip link set lo up

    echo "[6/10] Bringing up interfaces in 'fw'..."
    ip netns exec fw ip link set enp2s0 up
    ip netns exec fw ip link set enp4s0 up
    ip netns exec fw ip link set eno2 up
    ip netns exec fw ip link set veth-fw up
    ip netns exec fw ip link set lo up

    echo "[7/10] Enabling IP forwarding..."
    ip netns exec rtr sysctl -w net.ipv4.ip_forward=1
    ip netns exec fw sysctl -w net.ipv4.ip_forward=1

    echo "[8/10] Adding routes in 'rtr' to reach 9.0.0.0/24 and 9.0.1.0/24..."
    ip netns exec rtr ip route add 9.0.0.0/24 via 192.168.100.1 dev veth-rtr
    ip netns exec rtr ip route add 9.0.1.0/24 via 192.168.100.1 dev veth-rtr
    ip netns exec rtr ip route add 9.0.2.0/24 via 192.168.100.1 dev veth-rtr


    echo "[9/10] Adding routes in 'fw' to reach 10.0.2.0/24 and 10.0.3.0/24..."
    ip netns exec fw ip route add 10.0.2.0/24 via 192.168.100.2 dev veth-fw
    ip netns exec fw ip route add 10.0.3.0/24 via 192.168.100.2 dev veth-fw

    echo "[10/10] Setup complete."
    exit 0
fi

if [ "$1" = "clear" ]; then
    echo "[INFO] Clearing network namespaces..."

    # Cleanup fw namespace
    if ip netns list | grep -q fw; then
        echo "[1/4] Deleting veth-fw from 'fw'..."
        ip netns exec fw ip link set veth-fw down 2>/dev/null
        ip netns exec fw ip link delete veth-fw 2>/dev/null

        echo "[2/4] Moving physical interfaces from 'fw' back to root..."
        ip netns exec fw ip link set enp2s0 down 2>/dev/null
        ip netns exec fw ip link set enp4s0 down 2>/dev/null
        ip netns exec fw ip link set eno2 down 2>/dev/null
        ip netns exec fw ip link set enp2s0 netns 1 2>/dev/null
        ip netns exec fw ip link set enp4s0 netns 1 2>/dev/null
        ip netns exec fw ip link set eno2 netns 1 2>/dev/null

        ip netns del fw
    else
        echo "[WARN] Namespace 'fw' does not exist, skipping..."
    fi

    # Cleanup rtr namespace
    if ip netns list | grep -q rtr; then
```

```
104         echo "[3/4] Deleting veth-rtr from 'rtr'..."
105         ip netns exec rtr ip link set veth-rtr down 2>/dev/null
106         ip netns exec rtr ip link delete veth-rtr 2>/dev/null
107
108         echo "       Moving physical interfaces from 'rtr' back to root..."
109         ip netns exec rtr ip link set enp1s0f0 down 2>/dev/null
110         ip netns exec rtr ip link set enp1s0f1 down 2>/dev/null
111         ip netns exec rtr ip link set enp1s0f0 netns 1 2>/dev/null
112         ip netns exec rtr ip link set enp1s0f1 netns 1 2>/dev/null
113
114         ip netns del rtr
115     else
116         echo "[WARN] Namespace 'rtr' does not exist, skipping..."
117     fi
118
119     echo "[4/4] Cleanup complete. veth interfaces deleted, physical interfaces restored to root
        namespace."
120     exit 0
121 fi
122
123 echo "Invalid argument. Use 'create' or 'clear'."
124 exit 1
```

Listing A.1: Central-router setup script

A.2 iPerf3 campaigns

Table A.1: Per-second UDP throughput (iPerf3) during a 0-Byte ICMP-payload flood

Interval [s]	Transfer [MB]	Bitrate [Mbit/s]	Jitter	Lost/Total (Loss %)
0.00–1.00	107 MBytes	899 Mbits/s	0.015 ms	0/77726 (0%)
1.00–2.00	107 MBytes	900 Mbits/s	0.014 ms	0/77695 (0%)
2.00–3.00	107 MBytes	900 Mbits/s	0.014 ms	0/77690 (0%)
3.00–4.00	107 MBytes	900 Mbits/s	0.011 ms	0/77697 (0%)
4.00–5.00	107 MBytes	900 Mbits/s	0.014 ms	0/77690 (0%)
5.00–6.00	107 MBytes	900 Mbits/s	0.016 ms	0/77629 (0%)
6.00–7.00	107 MBytes	900 Mbits/s	0.013 ms	0/77755 (0%)
7.00–8.00	107 MBytes	900 Mbits/s	0.016 ms	0/77697 (0%)
8.00–9.00	107 MBytes	900 Mbits/s	0.015 ms	0/77692 (0%)
9.00–10.00	107 MBytes	900 Mbits/s	0.016 ms	0/77627 (0%)
10.00–11.00	107 MBytes	900 Mbits/s	0.016 ms	0/77760 (0%)
11.00–12.00	107 MBytes	900 Mbits/s	0.013 ms	0/77692 (0%)
12.00–13.00	107 MBytes	900 Mbits/s	0.015 ms	0/77697 (0%)

Continued on next page

Interval [s]	Transfer [MB]	Bitrate [Mbit/s]	Jitter	Lost/Total (Loss %)
13.00–14.00	107 MBytes	900 Mbits/s	0.015 ms	0/77692 (0%)
14.00–15.00	107 MBytes	900 Mbits/s	0.015 ms	0/77690 (0%)
15.00–16.00	107 MBytes	900 Mbits/s	0.015 ms	0/77695 (0%)
16.00–17.00	107 MBytes	900 Mbits/s	0.015 ms	0/77628 (0%)
17.00–18.00	107 MBytes	896 Mbits/s	0.013 ms	114/77511 (0.15%)
18.00–19.00	107 MBytes	900 Mbits/s	0.013 ms	0/77694 (0%)
19.00–20.00	107 MBytes	900 Mbits/s	0.013 ms	0/77698 (0%)
20.00–21.00	107 MBytes	900 Mbits/s	0.014 ms	0/77720 (0%)
21.00–22.00	107 MBytes	900 Mbits/s	0.014 ms	0/77696 (0%)
22.00–23.00	107 MBytes	900 Mbits/s	0.014 ms	0/77685 (0%)
23.00–24.00	107 MBytes	900 Mbits/s	0.016 ms	0/77701 (0%)
24.00–25.00	107 MBytes	900 Mbits/s	0.016 ms	0/77680 (0%)
25.00–26.00	107 MBytes	900 Mbits/s	0.021 ms	0/77701 (0%)
26.00–27.00	107 MBytes	900 Mbits/s	0.015 ms	0/77700 (0%)
27.00–28.00	107 MBytes	901 Mbits/s	0.017 ms	0/77804 (0%)
28.00–29.00	107 MBytes	900 Mbits/s	0.017 ms	0/77692 (0%)
29.00–30.00	107 MBytes	898 Mbits/s	0.018 ms	147/77694 (0.19%)
30.00–31.00	104 MBytes	871 Mbits/s	0.017 ms	2496/77693 (3.2%)
31.00–32.00	107 MBytes	900 Mbits/s	0.016 ms	0/77628 (0%)
32.00–33.00	104 MBytes	869 Mbits/s	0.018 ms	2639/77758 (3.4%)
33.00–34.00	107 MBytes	900 Mbits/s	0.019 ms	0/77695 (0%)
34.00–35.00	107 MBytes	900 Mbits/s	0.019 ms	0/77693 (0%)
35.00–36.00	107 MBytes	898 Mbits/s	0.018 ms	155/77694 (0.2%)
36.00–37.00	107 MBytes	897 Mbits/s	0.018 ms	275/77627 (0.35%)
37.00–38.00	107 MBytes	896 Mbits/s	0.017 ms	305/77759 (0.39%)
38.00–39.00	106 MBytes	886 Mbits/s	0.016 ms	1221/77691 (1.6%)
39.00–40.00	107 MBytes	900 Mbits/s	0.017 ms	0/77695 (0%)
40.00–41.00	107 MBytes	900 Mbits/s	0.016 ms	0/77693 (0%)
41.00–42.00	107 MBytes	900 Mbits/s	0.016 ms	0/77628 (0%)
42.00–43.00	107 MBytes	899 Mbits/s	0.014 ms	62/77760 (0.08%)
43.00–44.00	107 MBytes	900 Mbits/s	0.013 ms	0/77687 (0%)
44.00–45.00	107 MBytes	900 Mbits/s	0.016 ms	0/77633 (0%)
45.00–46.00	107 MBytes	900 Mbits/s	0.015 ms	0/77759 (0%)
46.00–47.00	107 MBytes	897 Mbits/s	0.017 ms	289/77694 (0.37%)

Continued on next page

Interval [s]	Transfer [MB]	Bitrate [Mbit/s]	Jitter	Lost/Total (Loss %)
47.00–48.00	107 MBytes	901 Mbits/s	0.017 ms	0/77797 (0%)
48.00–49.00	107 MBytes	900 Mbits/s	0.017 ms	0/77693 (0%)
49.00–50.00	107 MBytes	900 Mbits/s	0.013 ms	0/77629 (0%)
50.00–51.00	107 MBytes	900 Mbits/s	0.017 ms	0/77758 (0%)
51.00–52.00	107 MBytes	900 Mbits/s	0.020 ms	0/77693 (0%)
52.00–53.00	107 MBytes	900 Mbits/s	0.015 ms	0/77694 (0%)
53.00–54.00	107 MBytes	900 Mbits/s	0.015 ms	0/77690 (0%)
54.00–55.00	107 MBytes	900 Mbits/s	0.016 ms	0/77698 (0%)
55.00–56.00	107 MBytes	900 Mbits/s	0.013 ms	0/77694 (0%)
56.00–57.00	107 MBytes	900 Mbits/s	0.015 ms	0/77692 (0%)
57.00–58.00	107 MBytes	900 Mbits/s	0.014 ms	0/77694 (0%)
58.00–59.00	107 MBytes	900 Mbits/s	0.015 ms	0/77694 (0%)
59.00–60.00	107 MBytes	900 Mbits/s	0.014 ms	0/77690 (0%)
60.00–60.00	65.0 KBytes	606 Mbits/s	0.016 ms	0/46 (0%)

Table A.2: Per-second TCP throughput (iPerf3) during a 0-Byte ICMP-payload flood

Interval [s]	Transfer [MB]	Bitrate [Mbit/s]	Retr	Cwnd [KB]
0.00–1.00	113	949	13	373
1.00–2.00	112	937	2	373
2.00–3.00	112	935	5	373
3.00–4.00	112	937	3	373
4.00–5.00	112	938	3	373
5.00–6.00	111	934	0	373
6.00–7.00	112	938	0	373
7.00–8.00	111	932	0	373
8.00–9.00	112	940	0	373
9.00–10.00	112	940	0	431
10.00–11.00	111	934	0	431
11.00–12.00	111	934	0	431
12.00–13.00	112	943	0	431
13.00–14.00	111	934	0	431
14.00–15.00	112	935	0	431

Continued on next page

Interval [s]	Transfer [MB]	Bitrate [Mbit/s]	Retr	Cwnd [KB]
15.00–16.00	112	936	0	431
16.00–17.00	37.9	318	989	2.83
17.00–18.00	0.00	0.00	13	1.41
18.00–19.00	0.00	0.00	1	1.41
19.00–20.00	0.00	0.00	0	1.41
20.00–21.00	0.00	0.00	1	1.41
21.00–22.00	0.00	0.00	0	1.41
22.00–23.00	0.00	0.00	0	1.41
23.00–24.00	0.00	0.00	2	12.7
24.00–25.00	12.4	104	40	72.1
25.00–26.00	15.1	127	36	72.1
26.00–27.00	16.4	137	46	73.5
27.00–28.00	15.1	127	45	72.1
28.00–29.00	15.0	126	64	52.3
29.00–30.00	16.8	141	40	82.0
30.00–31.00	13.6	114	41	72.1
31.00–32.00	15.1	127	45	60.8
32.00–33.00	15.0	126	47	73.5
33.00–34.00	15.1	127	55	69.3
34.00–35.00	15.1	127	54	82.0
35.00–36.00	15.1	127	66	41.0
36.00–37.00	15.0	126	16	73.5
37.00–38.00	13.8	115	56	66.5
38.00–39.00	13.6	114	44	45.2
39.00–40.00	12.4	104	45	52.3
40.00–41.00	13.6	114	58	49.5
41.00–42.00	12.2	103	66	72.1
42.00–43.00	12.4	104	59	52.3
43.00–44.00	12.4	104	50	69.3
44.00–45.00	12.4	104	65	41.0
45.00–46.00	13.6	114	54	48.1
46.00–47.00	41.2	346	60	235
47.00–48.00	112	940	0	369
48.00–49.00	112	940	0	370

Continued on next page

Interval [s]	Transfer [MB]	Bitrate [Mbit/s]	Retr	Cwnd [KB]
49.00–50.00	112	940	0	370
50.00–51.00	112	937	0	372
51.00–52.00	111	929	0	372
52.00–53.00	112	938	0	397
53.00–54.00	111	933	0	406
54.00–55.00	112	940	0	406
55.00–56.00	112	941	0	406
56.00–57.00	111	930	0	406
57.00–58.00	112	940	0	406
58.00–59.00	112	942	0	406
59.00–60.00	111	929	0	406

Table A.3: Per-second UDP throughput (iPerf3) during a 1400-Byte ICMP-payload flood

Interval [s]	Transfer [MB]	Bitrate [Mbit/s]	Jitter	Lost/Total (Loss %)
0.00–1.00	107 MBytes	900 Mbits/s	0.015 ms	0/77732 (0%)
1.00–2.00	107 MBytes	900 Mbits/s	0.016 ms	0/77694 (0%)
2.00–3.00	107 MBytes	900 Mbits/s	0.016 ms	0/77697 (0%)
3.00–4.00	107 MBytes	900 Mbits/s	0.013 ms	0/77692 (0%)
4.00–5.00	107 MBytes	900 Mbits/s	0.015 ms	0/77691 (0%)
5.00–6.00	107 MBytes	900 Mbits/s	0.016 ms	0/77691 (0%)
6.00–7.00	107 MBytes	900 Mbits/s	0.014 ms	0/77695 (0%)
7.00–8.00	107 MBytes	900 Mbits/s	0.014 ms	0/77698 (0%)
8.00–9.00	107 MBytes	900 Mbits/s	0.015 ms	0/77689 (0%)
9.00–10.00	107 MBytes	900 Mbits/s	0.015 ms	0/77694 (0%)
10.00–11.00	107 MBytes	900 Mbits/s	0.015 ms	0/77694 (0%)
11.00–12.00	107 MBytes	900 Mbits/s	0.015 ms	0/77696 (0%)
12.00–13.00	107 MBytes	900 Mbits/s	0.014 ms	0/77691 (0%)
13.00–14.00	107 MBytes	900 Mbits/s	0.014 ms	0/77694 (0%)
14.00–15.00	107 MBytes	900 Mbits/s	0.016 ms	0/77693 (0%)
15.00–16.00	107 MBytes	900 Mbits/s	0.014 ms	0/77694 (0%)
16.00–17.00	75.8 MBytes	636 Mbits/s	0.032 ms	12427/67334 (18%)
17.00–18.00	57.9 MBytes	486 Mbits/s	0.070 ms	35798/77747 (46%)

Continued on next page

Interval [s]	Transfer [MB]	Bitrate [Mbit/s]	Jitter	Lost/Total (Loss %)
18.00–19.00	57.9 MBytes	486 Mbits/s	0.047 ms	35733/77684 (46%)
19.00–20.00	57.9 MBytes	486 Mbits/s	0.025 ms	35729/77682 (46%)
20.00–21.00	57.9 MBytes	486 Mbits/s	0.025 ms	35726/77673 (46%)
21.00–22.00	57.9 MBytes	486 Mbits/s	0.023 ms	35731/77676 (46%)
22.00–23.00	57.9 MBytes	486 Mbits/s	0.022 ms	35731/77682 (46%)
23.00–24.00	57.9 MBytes	486 Mbits/s	0.042 ms	35794/77744 (46%)
24.00–25.00	57.9 MBytes	486 Mbits/s	0.026 ms	35732/77679 (46%)
25.00–26.00	57.9 MBytes	486 Mbits/s	0.022 ms	35731/77680 (46%)
26.00–27.00	57.9 MBytes	486 Mbits/s	0.021 ms	35732/77679 (46%)
27.00–28.00	57.9 MBytes	486 Mbits/s	0.027 ms	35788/77737 (46%)
28.00–29.00	57.9 MBytes	486 Mbits/s	0.031 ms	35744/77690 (46%)
29.00–30.00	57.9 MBytes	486 Mbits/s	0.033 ms	35754/77703 (46%)
30.00–31.00	57.9 MBytes	486 Mbits/s	0.024 ms	35699/77646 (46%)
31.00–32.00	57.9 MBytes	486 Mbits/s	0.022 ms	35762/77713 (46%)
32.00–33.00	57.9 MBytes	486 Mbits/s	0.025 ms	35752/77699 (46%)
33.00–34.00	57.9 MBytes	486 Mbits/s	0.023 ms	35738/77687 (46%)
34.00–35.00	57.9 MBytes	486 Mbits/s	0.035 ms	35776/77726 (46%)
35.00–36.00	57.9 MBytes	486 Mbits/s	0.073 ms	35751/77698 (46%)
36.00–37.00	57.9 MBytes	486 Mbits/s	0.024 ms	35712/77663 (46%)
37.00–38.00	57.9 MBytes	486 Mbits/s	0.026 ms	35736/77683 (46%)
38.00–39.00	57.9 MBytes	486 Mbits/s	0.022 ms	35759/77703 (46%)
39.00–40.00	57.9 MBytes	486 Mbits/s	0.033 ms	35764/77711 (46%)
40.00–41.00	57.9 MBytes	486 Mbits/s	0.026 ms	35716/77663 (46%)
41.00–42.00	57.9 MBytes	486 Mbits/s	0.024 ms	35726/77672 (46%)
42.00–43.00	57.9 MBytes	486 Mbits/s	0.030 ms	35788/77735 (46%)
43.00–44.00	57.9 MBytes	486 Mbits/s	0.024 ms	35715/77661 (46%)
44.00–45.00	57.9 MBytes	486 Mbits/s	0.033 ms	35789/77733 (46%)
45.00–46.00	63.5 MBytes	533 Mbits/s	0.014 ms	33183/79186 (42%)
46.00–47.00	112 MBytes	937 Mbits/s	0.017 ms	5666/86528 (6.5%)
47.00–48.00	107 MBytes	900 Mbits/s	0.013 ms	0/77696 (0%)
48.00–49.00	107 MBytes	900 Mbits/s	0.015 ms	0/77692 (0%)
49.00–50.00	107 MBytes	900 Mbits/s	0.019 ms	0/77690 (0%)
50.00–51.00	107 MBytes	900 Mbits/s	0.013 ms	0/77695 (0%)
51.00–52.00	107 MBytes	900 Mbits/s	0.013 ms	0/77692 (0%)

Continued on next page

Interval [s]	Transfer [MB]	Bitrate [Mbit/s]	Jitter	Lost/Total (Loss %)
52.00–53.00	107 MBytes	900 Mbits/s	0.013 ms	0/77696 (0%)
53.00–54.00	107 MBytes	900 Mbits/s	0.012 ms	0/77693 (0%)
54.00–55.00	107 MBytes	900 Mbits/s	0.014 ms	0/77695 (0%)
55.00–56.00	107 MBytes	900 Mbits/s	0.014 ms	0/77691 (0%)
56.00–57.00	107 MBytes	900 Mbits/s	0.016 ms	0/77695 (0%)
57.00–58.00	107 MBytes	900 Mbits/s	0.014 ms	0/77697 (0%)
58.00–59.00	107 MBytes	900 Mbits/s	0.014 ms	0/77688 (0%)
59.00–60.00	107 MBytes	900 Mbits/s	0.018 ms	0/77696 (0%)
60.00–60.00	45.2 KBytes	510 Mbits/s	0.014 ms	0/32 (0%)

Table A.4: Per-second TCP throughput (iPerf3) during a 1400-Byte ICMP-payload flood

Interval [s]	Transfer [MB]	Bitrate [Mbit/s]	Retr	Cwnd [KB]
0.00–1.00	114	951	8	382
1.00–2.00	112	938	0	387
2.00–3.00	112	935	3	387
3.00–4.00	112	935	3	387
4.00–5.00	112	935	0	387
5.00–6.00	112	935	0	387
6.00–7.00	112	935	0	387
7.00–8.00	112	942	0	387
8.00–9.00	112	935	0	387
9.00–10.00	112	935	0	387
10.00–11.00	111	934	0	387
11.00–12.00	112	935	0	387
12.00–13.00	112	936	0	387
13.00–14.00	112	940	1	407
14.00–15.00	111	934	0	407
15.00–16.00	112	937	0	407
16.00–17.00	102	856	707	331
17.00–18.00	80.5	675	567	304

Continued on next page

Interval [s]	Transfer [MB]	Bitrate [Mbit/s]	Retr	Cwnd [KB]
18.00–19.00	82.6	693	550	317
19.00–20.00	86.0	721	763	324
20.00–21.00	85.5	717	992	324
21.00–22.00	81.6	685	1349	325
22.00–23.00	84.0	705	258	303
23.00–24.00	84.5	709	468	230
24.00–25.00	84.2	707	820	310
25.00–26.00	84.5	709	606	317
26.00–27.00	85.8	719	907	226
27.00–28.00	85.6	718	921	324
28.00–29.00	84.1	706	324	313
29.00–30.00	84.5	709	852	313
30.00–31.00	83.5	700	663	283
31.00–32.00	84.4	708	501	308
32.00–33.00	84.6	710	677	321
33.00–34.00	85.2	715	1257	321
34.00–35.00	85.0	713	988	321
35.00–36.00	85.9	720	658	321
36.00–37.00	83.5	700	990	263
37.00–38.00	83.9	704	270	293
38.00–39.00	84.9	712	387	310
39.00–40.00	84.2	707	857	226
40.00–41.00	85.4	716	1257	331
41.00–42.00	82.9	695	1230	339
42.00–43.00	84.5	709	1767	339
43.00–44.00	82.0	688	868	301
44.00–45.00	82.9	695	838	320
45.00–46.00	82.2	690	882	262
46.00–47.00	99.0	830	53	437
47.00–48.00	112	937	0	469
48.00–49.00	112	938	0	475
49.00–50.00	112	941	0	475

Continued on next page

Interval [s]	Transfer [MB]	Bitrate [Mbit/s]	Retr	Cwnd [KB]
50.00–51.00	111	931	0	505
51.00–52.00	112	936	0	509
52.00–53.00	112	936	0	518
53.00–54.00	112	940	0	518
54.00–55.00	112	940	0	518
55.00–56.00	111	930	0	518
56.00–57.00	112	940	0	518
57.00–58.00	112	940	0	518
58.00–59.00	111	930	0	518
59.00–60.00	112	939	0	518

A.3 VEREFOO input/output and translation in iptables rules

```
<?xml version="1.0" encoding="UTF-8"?>
<NFV
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../../xsd/
     nfvSchema.xsd">
  <graphs>
    <graph id="0">
      <!-- APs -->
      <node id="1" name="100.0.0.1">
        <!--  AP1  -->
        <neighbour name="10.0.3.254" />
        <neighbour name="9.0.0.1" />
        <neighbour name="9.0.1.1" />
        <neighbour name="9.0.2.1" />
      </node>
      <node id="2" name="100.0.0.2">
        <!--  AP2  -->
        <neighbour name="10.0.3.254" />
        <neighbour name="10.0.3.11" />
      </node>
      <node id="3" name="100.0.0.3">
        <!--  AP3  -->
        <neighbour name="10.0.3.254" />
        <neighbour name="10.0.3.21" />
      </node>
      <!-- FORWARDERs -->
      <node id="4" functional_type="FORWARDER" name="10.0.3.254">
        <neighbour name="100.0.0.1" />
        <neighbour name="100.0.0.2" />
        <neighbour name="100.0.0.3" />
        <neighbour name="10.0.3.3" />
      </node>
      <!-- WEBCLIENTs -->
      <node id="5" functional_type="WEBCLIENT" name="10.0.3.11">
        <neighbour name="100.0.0.2" />
```

```
      </node>
      <node id="6" functional_type="WEBCLIENT" name="10.0.3.21">
        <neighbour name="100.0.0.3" />
      </node>
      <node id="7" functional_type="WEBCLIENT" name="10.0.3.3">
        <neighbour name="10.0.3.254" />
      </node>
      <node id="8" functional_type="WEBCLIENT" name="9.0.0.1">
        <neighbour name="100.0.0.1" />
      </node>
      <node id="9" functional_type="WEBCLIENT" name="9.0.1.1">
        <neighbour name="100.0.0.1" />
      </node>
      <node id="10" functional_type="WEBCLIENT" name="9.0.2.1">
        <neighbour name="100.0.0.1" />
      </node>
    </graph>
  </graphs>
  <Constraints>
    <NodeConstraints />
    <LinkConstraints />
  </Constraints>
  <PropertyDefinition>
    <!-- rules for avoiding a pass-through in the upstream firewall -->
    <Property graph="0" name="IsolationProperty" src="9.0.-1.-1" dst="10.0.3.3" lv4proto="TCP"
     dst_port="0-79" />
    <Property graph="0" name="IsolationProperty" src="9.0.-1.-1" dst="10.0.3.11" lv4proto="TCP"
     dst_port="0-21" />
    <Property graph="0" name="IsolationProperty" src="9.0.-1.-1" dst="10.0.3.21" lv4proto="TCP"
     dst_port="0-21" />
    <!-- 10.0.3.11 -->
    <Property graph="0" name="ReachabilityProperty" src="9.0.-1.-1" dst="10.0.3.-1" lv4proto="TCP"
     dst_port="80" />
    <Property graph="0" name="ReachabilityProperty" src="9.0.-1.-1" dst="10.0.3.-1" lv4proto="TCP"
     dst_port="443" />
    <Property graph="0" name="ReachabilityProperty" src="9.0.-1.-1" dst="10.0.3.11" lv4proto="TCP"
     dst_port="22" />
    <Property graph="0" name="ReachabilityProperty" src="9.0.-1.-1" dst="10.0.3.11" lv4proto="TCP"
     dst_port="5432" />
    <!-- 10.0.3.21 -->
    <Property graph="0" name="ReachabilityProperty" src="9.0.-1.-1" dst="10.0.3.21" lv4proto="TCP"
     dst_port="22" />
    <Property graph="0" name="ReachabilityProperty" src="9.0.-1.-1" dst="10.0.3.21" lv4proto="TCP"
     dst_port="5432" />
    <!-- 10.0.3.3 -->
    <Property graph="0" name="IsolationProperty" src="9.0.-1.-1" dst="10.0.3.3" lv4proto="TCP"
     dst_port="5432" />
    <!-- custom script -->
    <Property graph="0" name="ReachabilityProperty" src="9.0.-1.-1" dst="10.0.3.-1" lv4proto="TCP"
     dst_port="5301" />
    <Property graph="0" name="ReachabilityProperty" src="9.0.-1.-1" dst="10.0.3.-1" lv4proto="UDP"
     dst_port="5301" />
    <!-- Block ICMP  -->
    <Property graph="0" name="IsolationProperty" src="9.0.-1.-1" dst="10.0.3.-1" lv4proto="OTHER"
     />
  </PropertyDefinition>
</NFV>
```

Listing A.2: XML provided as input to VEREFOO

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<NFV xsi:noNamespaceSchemaLocation="./xsd/nfvSchema.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

```
<graphs>
  <graph id="0">
    <node id="1" name="100.0.0.1" functional_type="FIREWALL">
      <neighbour name="10.0.3.254"/>
      <neighbour name="9.0.0.1"/>
      <neighbour name="9.0.1.1"/>
      <neighbour name="9.0.2.1"/>
      <configuration name="AutoConf">
        <firewall defaultAction="DENY">
          <elements>
            <source>9.0.0.1</source>
            <destination>10.0.3.21</destination>
            <protocol>TCP</protocol>
            <src_port>0-65535</src_port>
            <dst_port>5301</dst_port>
          </elements>
          <elements>
            <source>9.0.0.1</source>
            <destination>10.0.3.21</destination>
            <protocol>TCP</protocol>
            <src_port>0-65535</src_port>
            <dst_port>443</dst_port>
          </elements>
          <elements>
            <source>9.0.0.1</source>
            <destination>10.0.3.21</destination>
            <protocol>TCP</protocol>
            <src_port>0-65535</src_port>
            <dst_port>80</dst_port>
          </elements>
          <elements>
            <source>9.0.2.1</source>
            <destination>10.0.3.11</destination>
            <protocol>TCP</protocol>
            <src_port>0-65535</src_port>
            <dst_port>80</dst_port>
          </elements>
          <elements>
            <source>9.0.1.1</source>
            <destination>10.0.3.21</destination>
            <protocol>TCP</protocol>
            <src_port>0-65535</src_port>
            <dst_port>5301</dst_port>
          </elements>
          <elements>
            <source>9.0.1.1</source>
            <destination>10.0.3.11</destination>
            <protocol>TCP</protocol>
            <src_port>0-65535</src_port>
            <dst_port>5301</dst_port>
          </elements>
          <elements>
            <source>9.0.0.1</source>
            <destination>10.0.3.21</destination>
            <protocol>TCP</protocol>
            <src_port>0-65535</src_port>
            <dst_port>22</dst_port>
          </elements>
          <elements>
            <source>9.0.2.1</source>
            <destination>10.0.3.3</destination>
            <protocol>UDP</protocol>
            <src_port>0-65535</src_port>
```

```
67              <dst_port>5301</dst_port>
68            </elements>
69            <elements>
70              <source>9.0.0.1</source>
71              <destination>10.0.3.3</destination>
72              <protocol>UDP</protocol>
73              <src_port>0-65535</src_port>
74              <dst_port>5301</dst_port>
75            </elements>
76            <elements>
77              <source>9.0.2.1</source>
78              <destination>10.0.3.3</destination>
79              <protocol>TCP</protocol>
80              <src_port>0-65535</src_port>
81              <dst_port>5301</dst_port>
82            </elements>
83            <elements>
84              <source>9.0.1.1</source>
85              <destination>10.0.3.21</destination>
86              <protocol>UDP</protocol>
87              <src_port>0-65535</src_port>
88              <dst_port>5301</dst_port>
89            </elements>
90            <elements>
91              <source>9.0.1.1</source>
92              <destination>10.0.3.3</destination>
93              <protocol>TCP</protocol>
94              <src_port>0-65535</src_port>
95              <dst_port>80</dst_port>
96            </elements>
97            <elements>
98              <source>9.0.2.1</source>
99              <destination>10.0.3.11</destination>
100             <protocol>TCP</protocol>
101             <src_port>0-65535</src_port>
102             <dst_port>22</dst_port>
103           </elements>
104           <elements>
105             <source>9.0.0.1</source>
106             <destination>10.0.3.11</destination>
107             <protocol>TCP</protocol>
108             <src_port>0-65535</src_port>
109             <dst_port>5432</dst_port>
110           </elements>
111           <elements>
112             <source>9.0.0.1</source>
113             <destination>10.0.3.11</destination>
114             <protocol>TCP</protocol>
115             <src_port>0-65535</src_port>
116             <dst_port>80</dst_port>
117           </elements>
118           <elements>
119             <source>9.0.2.1</source>
120             <destination>10.0.3.21</destination>
121             <protocol>TCP</protocol>
122             <src_port>0-65535</src_port>
123             <dst_port>80</dst_port>
124           </elements>
125           <elements>
126             <source>9.0.2.1</source>
127             <destination>10.0.3.3</destination>
128             <protocol>TCP</protocol>
129             <src_port>0-65535</src_port>
```

```
130               <dst_port>80</dst_port>
131             </elements>
132             <elements>
133               <source>9.0.1.1</source>
134               <destination>10.0.3.11</destination>
135               <protocol>TCP</protocol>
136               <src_port>0-65535</src_port>
137               <dst_port>80</dst_port>
138             </elements>
139             <elements>
140               <source>9.0.0.1</source>
141               <destination>10.0.3.3</destination>
142               <protocol>TCP</protocol>
143               <src_port>0-65535</src_port>
144               <dst_port>5301</dst_port>
145             </elements>
146             <elements>
147               <source>9.0.1.1</source>
148               <destination>10.0.3.3</destination>
149               <protocol>TCP</protocol>
150               <src_port>0-65535</src_port>
151               <dst_port>5301</dst_port>
152             </elements>
153             <elements>
154               <source>9.0.1.1</source>
155               <destination>10.0.3.11</destination>
156               <protocol>TCP</protocol>
157               <src_port>0-65535</src_port>
158               <dst_port>5432</dst_port>
159             </elements>
160             <elements>
161               <source>9.0.2.1</source>
162               <destination>10.0.3.21</destination>
163               <protocol>TCP</protocol>
164               <src_port>0-65535</src_port>
165               <dst_port>22</dst_port>
166             </elements>
167             <elements>
168               <source>9.0.0.1</source>
169               <destination>10.0.3.3</destination>
170               <protocol>TCP</protocol>
171               <src_port>0-65535</src_port>
172               <dst_port>443</dst_port>
173             </elements>
174             <elements>
175               <source>9.0.1.1</source>
176               <destination>10.0.3.21</destination>
177               <protocol>TCP</protocol>
178               <src_port>0-65535</src_port>
179               <dst_port>5432</dst_port>
180             </elements>
181             <elements>
182               <source>9.0.2.1</source>
183               <destination>10.0.3.21</destination>
184               <protocol>UDP</protocol>
185               <src_port>0-65535</src_port>
186               <dst_port>5301</dst_port>
187             </elements>
188             <elements>
189               <source>9.0.1.1</source>
190               <destination>10.0.3.21</destination>
191               <protocol>TCP</protocol>
192               <src_port>0-65535</src_port>
```

```
193               <dst_port>443</dst_port>
194             </elements>
195             <elements>
196               <source>9.0.1.1</source>
197               <destination>10.0.3.11</destination>
198               <protocol>TCP</protocol>
199               <src_port>0-65535</src_port>
200               <dst_port>443</dst_port>
201             </elements>
202             <elements>
203               <source>9.0.0.1</source>
204               <destination>10.0.3.11</destination>
205               <protocol>TCP</protocol>
206               <src_port>0-65535</src_port>
207               <dst_port>5301</dst_port>
208             </elements>
209             <elements>
210               <source>9.0.2.1</source>
211               <destination>10.0.3.21</destination>
212               <protocol>TCP</protocol>
213               <src_port>0-65535</src_port>
214               <dst_port>5301</dst_port>
215             </elements>
216             <elements>
217               <source>9.0.0.1</source>
218               <destination>10.0.3.11</destination>
219               <protocol>TCP</protocol>
220               <src_port>0-65535</src_port>
221               <dst_port>443</dst_port>
222             </elements>
223             <elements>
224               <source>9.0.1.1</source>
225               <destination>10.0.3.3</destination>
226               <protocol>UDP</protocol>
227               <src_port>0-65535</src_port>
228               <dst_port>5301</dst_port>
229             </elements>
230             <elements>
231               <source>9.0.2.1</source>
232               <destination>10.0.3.11</destination>
233               <protocol>TCP</protocol>
234               <src_port>0-65535</src_port>
235               <dst_port>5432</dst_port>
236             </elements>
237             <elements>
238               <source>9.0.2.1</source>
239               <destination>10.0.3.3</destination>
240               <protocol>TCP</protocol>
241               <src_port>0-65535</src_port>
242               <dst_port>443</dst_port>
243             </elements>
244             <elements>
245               <source>9.0.2.1</source>
246               <destination>10.0.3.21</destination>
247               <protocol>TCP</protocol>
248               <src_port>0-65535</src_port>
249               <dst_port>5432</dst_port>
250             </elements>
251             <elements>
252               <source>9.0.0.1</source>
253               <destination>10.0.3.3</destination>
254               <protocol>TCP</protocol>
255               <src_port>0-65535</src_port>
```

```
256               <dst_port>80</dst_port>
257             </elements>
258             <elements>
259               <source>9.0.0.1</source>
260               <destination>10.0.3.11</destination>
261               <protocol>UDP</protocol>
262               <src_port>0-65535</src_port>
263               <dst_port>5301</dst_port>
264             </elements>
265             <elements>
266               <source>9.0.1.1</source>
267               <destination>10.0.3.21</destination>
268               <protocol>TCP</protocol>
269               <src_port>0-65535</src_port>
270               <dst_port>80</dst_port>
271             </elements>
272             <elements>
273               <source>9.0.2.1</source>
274               <destination>10.0.3.11</destination>
275               <protocol>TCP</protocol>
276               <src_port>0-65535</src_port>
277               <dst_port>5301</dst_port>
278             </elements>
279             <elements>
280               <source>9.0.1.1</source>
281               <destination>10.0.3.11</destination>
282               <protocol>TCP</protocol>
283               <src_port>0-65535</src_port>
284               <dst_port>22</dst_port>
285             </elements>
286             <elements>
287               <source>9.0.1.1</source>
288               <destination>10.0.3.21</destination>
289               <protocol>TCP</protocol>
290               <src_port>0-65535</src_port>
291               <dst_port>22</dst_port>
292             </elements>
293             <elements>
294               <source>9.0.2.1</source>
295               <destination>10.0.3.11</destination>
296               <protocol>UDP</protocol>
297               <src_port>0-65535</src_port>
298               <dst_port>5301</dst_port>
299             </elements>
300             <elements>
301               <source>9.0.0.1</source>
302               <destination>10.0.3.11</destination>
303               <protocol>TCP</protocol>
304               <src_port>0-65535</src_port>
305               <dst_port>22</dst_port>
306             </elements>
307             <elements>
308               <source>9.0.1.1</source>
309               <destination>10.0.3.3</destination>
310               <protocol>TCP</protocol>
311               <src_port>0-65535</src_port>
312               <dst_port>443</dst_port>
313             </elements>
314             <elements>
315               <source>9.0.2.1</source>
316               <destination>10.0.3.21</destination>
317               <protocol>TCP</protocol>
318               <src_port>0-65535</src_port>
```

```
319               <dst_port>443</dst_port>
320             </elements>
321             <elements>
322               <source>9.0.2.1</source>
323               <destination>10.0.3.11</destination>
324               <protocol>TCP</protocol>
325               <src_port>0-65535</src_port>
326               <dst_port>443</dst_port>
327             </elements>
328             <elements>
329               <source>9.0.1.1</source>
330               <destination>10.0.3.11</destination>
331               <protocol>UDP</protocol>
332               <src_port>0-65535</src_port>
333               <dst_port>5301</dst_port>
334             </elements>
335             <elements>
336               <source>9.0.0.1</source>
337               <destination>10.0.3.21</destination>
338               <protocol>UDP</protocol>
339               <src_port>0-65535</src_port>
340               <dst_port>5301</dst_port>
341             </elements>
342             <elements>
343               <source>9.0.0.1</source>
344               <destination>10.0.3.21</destination>
345               <protocol>TCP</protocol>
346               <src_port>0-65535</src_port>
347               <dst_port>5432</dst_port>
348             </elements>
349           </firewall>
350         </configuration>
351       </node>
352       <node id="2" name="100.0.0.2" functional_type="FORWARDER">
353         <neighbour name="10.0.3.254"/>
354         <neighbour name="10.0.3.11"/>
355         <configuration name="ForwardConf">
356           <forwarder>
357             <name>Forwarder</name>
358           </forwarder>
359         </configuration>
360       </node>
361       <node id="3" name="100.0.0.3" functional_type="FORWARDER">
362         <neighbour name="10.0.3.254"/>
363         <neighbour name="10.0.3.21"/>
364         <configuration name="ForwardConf">
365           <forwarder>
366             <name>Forwarder</name>
367           </forwarder>
368         </configuration>
369       </node>
370       <node id="4" name="10.0.3.254" functional_type="FORWARDER">
371         <neighbour name="100.0.0.1"/>
372         <neighbour name="100.0.0.2"/>
373         <neighbour name="100.0.0.3"/>
374         <neighbour name="10.0.3.3"/>
375       </node>
376       <node id="5" name="10.0.3.11" functional_type="WEBCLIENT">
377         <neighbour name="100.0.0.2"/>
378       </node>
379       <node id="6" name="10.0.3.21" functional_type="WEBCLIENT">
380         <neighbour name="100.0.0.3"/>
381       </node>
```

```
      <node id="7" name="10.0.3.3" functional_type="WEBCLIENT">
        <neighbour name="10.0.3.254"/>
      </node>
      <node id="8" name="9.0.0.1" functional_type="WEBCLIENT">
        <neighbour name="100.0.0.1"/>
      </node>
      <node id="9" name="9.0.1.1" functional_type="WEBCLIENT">
        <neighbour name="100.0.0.1"/>
      </node>
      <node id="10" name="9.0.2.1" functional_type="WEBCLIENT">
        <neighbour name="100.0.0.1"/>
      </node>
    </graph>
  </graphs>
  <Constraints>
    <NodeConstraints/>
    <LinkConstraints/>
  </Constraints>
  <PropertyDefinition>
        ...
        </PropertyDefinition>

</NFV>
```

Listing A.3: XML result produced by VEREFOO

```
#!/bin/sh
cmd="sudo ip netns exec fw iptables"
${cmd} -F
${cmd} -P INPUT DROP
${cmd} -P FORWARD DROP
${cmd} -P OUTPUT DROP
${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 5301 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.0.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 443 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.0.1/32 --sport 443 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 80 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.0.1/32 --sport 80 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 80 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.2.1/32 --sport 80 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 5301 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.1.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 5301 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.1.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 22 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.0.1/32 --sport 22 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p udp -s 9.0.2.1/32 -d 10.0.3.3/32 --sport 0:65535 --dport 5301 -j ACCEPT
${cmd} -A FORWARD -p udp -s 10.0.3.3/32 -d 9.0.2.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p udp -s 9.0.0.1/32 -d 10.0.3.3/32 --sport 0:65535 --dport 5301 -j ACCEPT
${cmd} -A FORWARD -p udp -s 10.0.3.3/32 -d 9.0.0.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.3/32 --sport 0:65535 --dport 5301 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.3.3/32 -d 9.0.2.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p udp -s 9.0.1.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 5301 -j ACCEPT
${cmd} -A FORWARD -p udp -s 10.0.3.21/32 -d 9.0.1.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.3/32 --sport 0:65535 --dport 80 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.3.3/32 -d 9.0.1.1/32 --sport 80 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 22 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.2.1/32 --sport 22 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 5432 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.0.1/32 --sport 5432 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 80 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.0.1/32 --sport 80 --dport 0:65535 -j ACCEPT
${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 80 -j ACCEPT
```

```
38 ${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.2.1/32 --sport 80 --dport 0:65535 -j ACCEPT
39 ${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.3/32 --sport 0:65535 --dport 80 -j ACCEPT
40 ${cmd} -A FORWARD -p tcp -s 10.0.3.3/32 -d 9.0.2.1/32 --sport 80 --dport 0:65535 -j ACCEPT
41 ${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 80 -j ACCEPT
42 ${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.1.1/32 --sport 80 --dport 0:65535 -j ACCEPT
43 ${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.3/32 --sport 0:65535 --dport 5301 -j ACCEPT
44 ${cmd} -A FORWARD -p tcp -s 10.0.3.3/32 -d 9.0.0.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
45 ${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.3/32 --sport 0:65535 --dport 5301 -j ACCEPT
46 ${cmd} -A FORWARD -p tcp -s 10.0.3.3/32 -d 9.0.1.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
47 ${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 5432 -j ACCEPT
48 ${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.1.1/32 --sport 5432 --dport 0:65535 -j ACCEPT
49 ${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 22 -j ACCEPT
50 ${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.2.1/32 --sport 22 --dport 0:65535 -j ACCEPT
51 ${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.3/32 --sport 0:65535 --dport 443 -j ACCEPT
52 ${cmd} -A FORWARD -p tcp -s 10.0.3.3/32 -d 9.0.0.1/32 --sport 443 --dport 0:65535 -j ACCEPT
53 ${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 5432 -j ACCEPT
54 ${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.1.1/32 --sport 5432 --dport 0:65535 -j ACCEPT
55 ${cmd} -A FORWARD -p udp -s 9.0.2.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 5301 -j ACCEPT
56 ${cmd} -A FORWARD -p udp -s 10.0.3.21/32 -d 9.0.2.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
57 ${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 443 -j ACCEPT
58 ${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.1.1/32 --sport 443 --dport 0:65535 -j ACCEPT
59 ${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 443 -j ACCEPT
60 ${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.1.1/32 --sport 443 --dport 0:65535 -j ACCEPT
61 ${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 5301 -j ACCEPT
62 ${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.0.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
63 ${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 5301 -j ACCEPT
64 ${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.2.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
65 ${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 443 -j ACCEPT
66 ${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.0.1/32 --sport 443 --dport 0:65535 -j ACCEPT
67 ${cmd} -A FORWARD -p udp -s 9.0.1.1/32 -d 10.0.3.3/32 --sport 0:65535 --dport 5301 -j ACCEPT
68 ${cmd} -A FORWARD -p udp -s 10.0.3.3/32 -d 9.0.1.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
69 ${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 5432 -j ACCEPT
70 ${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.2.1/32 --sport 5432 --dport 0:65535 -j ACCEPT
71 ${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.3/32 --sport 0:65535 --dport 443 -j ACCEPT
72 ${cmd} -A FORWARD -p tcp -s 10.0.3.3/32 -d 9.0.2.1/32 --sport 443 --dport 0:65535 -j ACCEPT
73 ${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 5432 -j ACCEPT
74 ${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.2.1/32 --sport 5432 --dport 0:65535 -j ACCEPT
75 ${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.3/32 --sport 0:65535 --dport 80 -j ACCEPT
76 ${cmd} -A FORWARD -p tcp -s 10.0.3.3/32 -d 9.0.0.1/32 --sport 80 --dport 0:65535 -j ACCEPT
77 ${cmd} -A FORWARD -p udp -s 9.0.0.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 5301 -j ACCEPT
78 ${cmd} -A FORWARD -p udp -s 10.0.3.11/32 -d 9.0.0.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
79 ${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 80 -j ACCEPT
80 ${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.1.1/32 --sport 80 --dport 0:65535 -j ACCEPT
81 ${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 5301 -j ACCEPT
82 ${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.2.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
83 ${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 22 -j ACCEPT
84 ${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.1.1/32 --sport 22 --dport 0:65535 -j ACCEPT
85 ${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 22 -j ACCEPT
86 ${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.1.1/32 --sport 22 --dport 0:65535 -j ACCEPT
87 ${cmd} -A FORWARD -p udp -s 9.0.2.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 5301 -j ACCEPT
88 ${cmd} -A FORWARD -p udp -s 10.0.3.11/32 -d 9.0.2.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
89 ${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 22 -j ACCEPT
90 ${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.0.1/32 --sport 22 --dport 0:65535 -j ACCEPT
91 ${cmd} -A FORWARD -p tcp -s 9.0.1.1/32 -d 10.0.3.3/32 --sport 0:65535 --dport 443 -j ACCEPT
92 ${cmd} -A FORWARD -p tcp -s 10.0.3.3/32 -d 9.0.1.1/32 --sport 443 --dport 0:65535 -j ACCEPT
93 ${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 443 -j ACCEPT
94 ${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.2.1/32 --sport 443 --dport 0:65535 -j ACCEPT
95 ${cmd} -A FORWARD -p tcp -s 9.0.2.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 443 -j ACCEPT
96 ${cmd} -A FORWARD -p tcp -s 10.0.3.11/32 -d 9.0.2.1/32 --sport 443 --dport 0:65535 -j ACCEPT
97 ${cmd} -A FORWARD -p udp -s 9.0.1.1/32 -d 10.0.3.11/32 --sport 0:65535 --dport 5301 -j ACCEPT
98 ${cmd} -A FORWARD -p udp -s 10.0.3.11/32 -d 9.0.1.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
99 ${cmd} -A FORWARD -p udp -s 9.0.0.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 5301 -j ACCEPT
100 ${cmd} -A FORWARD -p udp -s 10.0.3.21/32 -d 9.0.0.1/32 --sport 5301 --dport 0:65535 -j ACCEPT
```

```
101 ${cmd} -A FORWARD -p tcp -s 9.0.0.1/32 -d 10.0.3.21/32 --sport 0:65535 --dport 5432 -j ACCEPT
102 ${cmd} -A FORWARD -p tcp -s 10.0.3.21/32 -d 9.0.0.1/32 --sport 5432 --dport 0:65535 -j ACCEPT
```

Listing A.4: iptables translation

Bibliography

[1] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated Firewall Configuration in Virtual Networks", *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1559–1576, Mar. 2023, ISSN: 1545-5971, 1941-0018, 2160-9209. DOI: 10.1109/TDSC.2022.3160293. [Online]. Available: https://ieeexplore.ieee.org/document/9737389/ (visited on 08/02/2025).

[2] L. Durante, L. Seno, and A. Valenzano, "A Formal Model and Technique to Redistribute the Packet Filtering Load in Multiple Firewall Networks", *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 2637–2651, 2021, ISSN: 1556-6013, 1556-6021. DOI: 10.1109/TIFS.2021.3057552. [Online]. Available: https://ieeexplore.ieee.org/document/9350284/ (visited on 08/02/2025).

[3] *Iptables(8) - Linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man8/iptables.8.html (visited on 08/03/2025).

[4] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated optimal firewall orchestration and configuration in virtualized networks", in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, Budapest, Hungary: IEEE, Apr. 2020, pp. 1–7, ISBN: 978-1-7281-4973-8. DOI: 10.1109/NOMS47738.2020.9110402. [Online]. Available: https://ieeexplore.ieee.org/document/9110402/ (visited on 08/02/2025).

[5] *Iptables-save(8) - Linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man8/iptables-save.8.html (visited on 08/06/2025).

[6] *Iptables-restore(8) - Linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man8/iptables-restore.8.html (visited on 08/06/2025).

[7] J. Engelhardt, *Schematic for the packet flow paths through Linux networking and Xtables*, May 2019. [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=8575254 (visited on 08/07/2025).

[8] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver", en, in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer, 2008, pp. 337–340, ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[9] *Z3Prover/z3*, original-date: 2015-03-26T18:16:07Z, Aug. 2025. [Online]. Available: https://github.com/Z3Prover/z3 (visited on 08/02/2025).

[10] S. Singh, Y.-S. Jeong, and J. H. Park, "A survey on cloud computing security: Issues, threats, and solutions", en, *Journal of Network and Computer Applications*, vol. 75, pp. 200–222, Nov. 2016, ISSN: 10848045. DOI: 10.1016/j.jnca.2016.09.002. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1084804516301990 (visited on 08/14/2025).

[11] D. Bringhenti, R. Sisto, and F. Valenza, "A demonstration of VEREFOO: An automated framework for virtual firewall configuration", in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, Madrid, Spain: IEEE, Jun. 2023, pp. 293–295, ISBN: 979-8-3503-9980-6. DOI: 10.1109/NetSoft57336.2023.10175442. [Online]. Available: https://ieeexplore.ieee.org/document/10175442/ (visited on 08/03/2025).

[12] *iPerf - iPerf3 and iPerf2 user documentation*. [Online]. Available: https://iperf.fr/iperf-doc.php#3doc (visited on 08/06/2025).

[13] *Iperf(1): Perform network throughput tests - Linux man page*. [Online]. Available: https://linux.die.net/man/1/iperf (visited on 08/03/2025).

[14] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems", *IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008)*, pp. 1–499, Jun. 2020. DOI: 10.1109/IEEESTD.2020.9120376. [Online]. Available: https://ieeexplore.ieee.org/document/9120376 (visited on 08/11/2025).

[15] *Tcpdump(1) man page | TCPDUMP & LIBPCAP*. [Online]. Available: https://www.tcpdump.org/manpages/tcpdump.1.html (visited on 08/03/2025).

[16] *Tshark(1)*. [Online]. Available: https://www.wireshark.org/docs/man-pages/tshark.html (visited on 08/07/2025).

[17] *Pcap-tstamp(7) — Arch manual pages*. [Online]. Available: https://man.archlinux.org/man/pcap-tstamp.7.en (visited on 08/11/2025).

[18] *Struct sk_ buff — The Linux Kernel documentation*. [Online]. Available: https://docs.kernel.org/networking/skbuff.html (visited on 08/11/2025).

[19] *Timestamping — The Linux Kernel documentation*. [Online]. Available: https://docs.kernel.org/networking/timestamping.html (visited on 08/11/2025).

[20] *Kernel.org/doc/Documentation/networking/timestamping.txt*. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/timestamping.txt (visited on 08/11/2025).

[21] *Hping3 | Kali Linux Tools*, English. [Online]. Available: https://www.kali.org/tools/hping3/ (visited on 08/03/2025).

[22] *Hping3(8) - Linux man page*. [Online]. Available: https://linux.die.net/man/8/hping3 (visited on 08/03/2025).

[23] *Ip-netns(8) - Linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man8/ip-netns.8.html (visited on 08/11/2025).

[24] *Tmpfs*, en, Page Version ID: 1281546210, Mar. 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Tmpfs&oldid=1281546210 (visited on 08/03/2025).

[25] L. Zhou, M. Liao, C. Yuan, and H. Zhang, "Low-Rate DDoS Attack Detection Using Expectation of Packet Size", en, *Security and Communication Networks*, vol. 2017, pp. 1–14, 2017, ISSN: 1939-0114, 1939-0122. DOI: 10.1155/2017/3691629. [Online]. Available: https://www.hindawi.com/journals/scn/2017/3691629/ (visited on 09/02/2025).

[26] F. Pizzato, D. Bringhenti, R. Sisto, and F. Valenza, "Automatic and optimized firewall reconfiguration", in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, Seoul, Korea, Republic of: IEEE, May 2024, pp. 1–9, ISBN: 979-8-3503-2793-9. DOI: 10.1109/NOMS59830.2024.10575212. [Online]. Available: https://ieeexplore.ieee.org/document/10575212/ (visited on 08/02/2025).

[27] D. Bringhenti, F. Pizzato, R. Sisto, and F. Valenza, "A Looping Process for Cyberattack Mitigation", in *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*, London, United Kingdom: IEEE, Sep. 2024, pp. 276–281, ISBN: 979-8-3503-7536-7. DOI: 10.1109/CSR61664.2024.10679501. [Online]. Available: https://ieeexplore.ieee.org/document/10679501/ (visited on 08/02/2025).